VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Bakalářská práce

Technologie XNA

The XNA Framework

2012 Stanislav Velký

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Stanislav Velký**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Technologie XNA
The XNA Framework

Zásady pro vypracování:

Cílem práce je detailně prozkoumat technologii XNA a porovnat ji s klasickým přístupem k vývoji grafických aplikací zejména pomocí DirectX. Práce bude obsahovat následující body.

1. Popište možnosti a směr vývoje XNA.
2. Srovnejte XNA s jinými technologiemi zaměřenými na stejnou oblast.
3. Porovnejte použití XNA s vývojem grafických aplikací přímo pod Direct3D a OpenGL.
4. Vypracujte ukázkovou aplikaci prezentující možnosti XNA v kontextu herního engine nebo profesionální aplikace.
5. Shrňte přínosy a možná úskalí spojené s XNA.

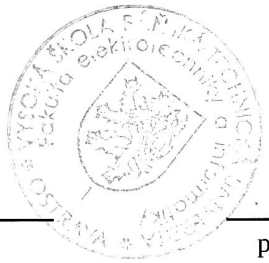
Seznam doporučené odborné literatury:

[1] Jaegers, K. XNA 4.0 Game Development by Example: Beginner's Guide. 2010. 428 s. ISBN 978-1849690669.
[2] Reed, A. Learning XNA 4.0. 2010. 544 s. ISBN 978-1449394622.
[3] Gregory, J. Game Engine Architecture. 2009. 864 s. ISBN 978-1568814131.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Tomáš Fabián**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 24. dubna 2012 ..... Velky!.....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 24. dubna 2012 ..... Velky!.....

Tímto bych chtěl poděkovat vedoucímu bakalářské práce Ing. Tomášovi Fabiánovi za cenné rady, připomínky a metodické vedení práce.

Abstrakt

Tato práce se zabývá průzkumem frameworku XNA. Cílem bylo porovnat XNA s klasickým programováním her pomocí DirectX a dalšími podobnými technologiemi s XNA. V této práci je také popsán způsob tvorby jednoduché hry s využitím fyzikálního enginu.

Klíčová slova: XNA, C#, SlimDx, DirectX, PhysX, bakalářská práce, shadow map

Abstract

This work deals with the exploration of XNA framework. The aim was to compare XNA with the classical programming game with DirectX and other similar technologies with XNA. This work also describes, how creating simple games using physical engine.

Keywords: XNA, C#, SlimDx, DirectX, PhysX, bachelor thesis, shadow map

Seznam použitých zkratek a symbolů

XNA	–	XNA's not Acronymed
XACT	–	Cross-platform Audio Creation Tool
C#	–	C-Sharp
HLSL	–	High Level Shader Language
CPU	–	Central processing unit
GPU	–	Graphics processing unit
PPU	–	Physics processing unit

Obsah

Seznam obrázků

1 Úvod

Počítačové hry se v průběhu několika let hodně změnily. Grafika stejně jako fyzika ve hrách se neustále blíží reálnému světu. Za tímto vývojem stojí mnoho práce a úsilí, vyvíjejí se nové techniky, které se snaží ušetřit práci procesoru nebo grafické karty a zároveň vypadají lépe. Ještě před několika lety jsme museli vyvinout velkou snahu, abychom vůbec nějakou hru vytvořily. Dnes nám pomáhají různé nástroje. Tyto nástroje se nám snaží ušetřit čas a práci s nízkoúrovňovými věcmi, které nám mohou způsobit problémy. Zároveň jsme schopni hru vyrobit s méně početným týmem. Při tomto vývoji jsme ovšem omezování a nelze využívat všechny možnosti, díky kterým bychom mohli vyvinout konkurenceschopnou hru. Pro OpenGL a DirectX existuje celá řada frameworků nebo enginů, které můžeme využívat. Mezi nejznámější a zároveň jeden z velmi podporovaných frameworků je XNA, který je nástupce již ukončeného Managed DirectX.

V této práci jsem se pokusil o prozkoumání tohoto frameworku a snažil jsem se najít takové věci, které nám mohou při vytváření her s tímto frameworkem pomoci nebo naopak v čem nám tuto práci může ztížit. Práce je rozdělena do několika částí. V první části přibližuji možnosti, které framework XNA nabízí. Další část je praktická, kde popisuji jednotlivé části výsledné hry. Třetí kapitola se zaměřuje na fyzikální engine a jeho začlenění do XNA. V poslední kapitole porovnávám jednotlivé možnosti různých nástrojů, se kterými lze vytvořit hru. V rámci práce jsem vytvořil také jednoduchou hru.

1.1 Engine

Engine je nástroj sloužící k vytváření her. Poskytuje herním vývojářům komponenty k vývoji her na různá cílová zařízení s minimálním zásahem do zdrojového kódu. Cílem herního enginu je zajistit herním tvůrcům dostatečně širokou paletu nástrojů a funkcí, s jejichž pomocí by byli schopni vyvíjet hry jednoduše a v co nejvyšší kvalitě. Mezi hlavní komponenty patřící do enginu a zajišťující hře reálnost patří načítání, zobrazování a animování modelů, fyzika, detekce kolizí mezi objekty, vstup, umělá inteligence, multiplayer a mnoho dalších.

Základní komponenty enginu:

- *Graficky modul* je zodpovědný za zobrazování modelů, textur ve hře, vykreslování různých efektů jako je voda, výbuchy, mlha atd.
- *Fyzikální komponenta* je zodpovědná za fyziku, která má za cíl simulovat reálný svět. Snaží se simulovat pohyb jak pevných, tak měkkých (vlasy, svaly, tuk) těles, kolize, simulace tekutin, kouře, výbuchy. V posledních letech je tato simulace prováděna na speciálních nebo grafických kartách.
- *Zvukový modul* odpovídá za vytváření zvuků (zvukových efektů), který pomáhá lokalizovat objekty ve hře, vytvářet zvuk v různých prostředí jako jsou jeskyně, koupelny nebo zvuk pod vodou.

- *Komponenta zajišťující vstup* odpovídá za zpracovávání vstupních zařízení jako jsou klávesnice, gamepady, joysticky a další.
- *Skriptování* poskytuje možnost pro jednoduché programování kódu hry. Každý engine podporuje různé skriptovací jazyky např. Python nebo Lua. Vznikají také skriptovací jazyky, které podporují jen konkrétní herní enginy např. UnrealScript.

Před vznikem herních enginů hry vznikaly většinou od nuly a kód se musel pokaždé vytvářet znovu. Za první engine, který kdy vznikl je považován *Freescape*, vyvinut v roce 1986, používaný pro nekomerční účely a díky úspěchu byl portován na *ZX Spectrum, IBM PC, Commodore 64, Commodore Amiga a Atari ST*. První hra bežící na tomto engine byla Driller. Největší zlom vývoje herních enginu však nastal v polovině 90. let, kdy společnost *id Software* vytvořila hru *Doom*, která se stala velmi populární. Později začala *id Software* vydávat hry, které byly založeny právě na hře *Doom*, tím vznikl herní engine, který postupem času byl řadu let vylepšován. Časem začalo vznikat neuvěřitelné množství enginu, které nabízejí různé editory pro rychlejší vývoj a další věci ulehčující práci. Herní enginy lze roztřídit podle toho jaký typ her podporují.

First-person shooter

Hry z pohledu první osoby. Tyto hry od roku 1973, kdy vznikla první first-person shooter hra *Maze War*, kde jsme se mohli pohybovat bludištěm až po dnešní hry typu *FarCry* prodělaly mnoho změn jak po grafické, tak i po fyzikální stránce.

Obrázek 1: Ukázka grafiky od hry Maze War až po Far Cry 3

- **id Tech** vyvinutý společností *id Software*, je jedním z prvních first-person shooter enginů. Nejprve byl známy pod názvem *Doom engine*, vývojářem byl John Carmack. Později byl však přejmenován na *id Tech*. Dosud vzniklo šest verzí těchto enginů a je jedním z prvních, kde byl využit model client-server.

  Hry: Doom 1-3, Quake 1-4, Medal of Honor

- **Unreal Engine** [1] vyvinutý společností *Epic Games*. Jádro je napsané v jazyku C++, ale jsou zde využity i jiné jazyky jako C#, HLSL, GLSL, CG nebo UnrealScript. Vývoj začal v roce 1998, kdy vznikla první verze *Unreal Engine 1*, která měla za úkol provádět vykreslování, detekci kolizi, umělou inteligenci, multiplayer. V *Unreal Engine 2* bylo kompletně přepsáno jádro a přibyly nové cílové platformy (Xbox). Verze

Unreal Engine 3 nabídla nové techniky grafického zobrazení HDRR, per-pixel osvětlení a dynamické stíny. Zatím je poslední verze *Unreal Engine 4*, která se více zaměřuje na konzole.

Hry: Mass Effect, Unreal Tournament, Gears of War

- **CryEngine** [2] vyvinutý společností *Crytek*, byl představen jako technologické demo pro *NVIDIA*, ale když v něm společnost viděla potenciál, byla z něj vytvořena hra.

  Hry: FarCry, Crysis

Massively multiplayer online game

Dnes stále více oblíbené hry, kde enginy jsou vyvinuté speciálně pro multiplayer a musí zajistit kvalitní podporu serverů.

- **BigWorld** [3] engine založený Australskou společností s podporou pro počítače s Windows, iOS nebo zařízení Xbox360 a Playstation 3. Server je napsán pod operačním systémem Linux, který dokáže obsloužit až 250 000 hráčů.

  Hry: World of Tanks, World of Warplanes, World of Battleships

- **Exit Games** je Německá společnost vydávající MMO engine na cílové platformy počítače, mobily a konzole. Server je napsán v C#/.Net, který v závislosti na licencích dokáže obsloužit různý počet uživatelů.

  Hry: F1 Online: The Game

Strategie

Hry, kde je kladen velký důraz na promyšlení a naplánování jednotlivých akcí, vedoucích k vítězství. Strategie můžeme rozdělit do několika druhů podle způsobu jejího hraní na realtimové nebo tahové strategie.

Realtimová strategie

Realtimová strategie je podžánrem strategických počítačových her, kde akce ve hře jsou kontinuální a hráč je nucen měnit své rozhodnutí v závislosti s měnícím se stavem hry. Tento typ her zaznamenal velký pokrok od hry *Dune*, která je považována za první realtimovou strategií, až po sérii *Command & Conquer*.

Obrázek 2: Ukázka grafiky od hry Dune 2 až po Command & Conquer 3

- **Sage** engine (Westwood 3D) byl založen a silně modifikován z enginu *SurRender 3D* založeným *Hybrid Graphics*. Později byl zakoupen společností *Electronic Arts* a přejmenován na *SAGE*, kde vykreslovací jádro zůstalo nezměněno, ovšem další části již byly přepracovány.

  Hry: Command & Conquer: Generals, Command & Conquer: Red Alert 3

- **Spring** engine vytvořený Švédskou společností *Yankspankers*, byl vytvořen ze hry *Total Annihilation*. Nabízí možnost vyvíjet hry jako multi-player pomocí jazyku Lua.

  Hry: Balanced Annihilation

Tahové strategie

Tahové strategie na rozdíl od realtimových strategií, kde všichni hráči hrají současně, se hráči při hraní střídají. Tento typ hry zaznamenal také velký pokrok od hry *UFO: Enemy Unknown*, která byla považovaná za nejlepší hru svého druhu až po sérii *Heroes of Might and Magic*.

Obrázek 3: Ukázka grafiky od hry UFO: Enemy Unknown až po Heroes of Might and Magic V

- **Silent Storm** engine vyvinutý společností *Nival Interactive* pro hru *Silent Storm*. Tento engine nabízí propracovanou fyziku, kde lze téměř všechno zničit, ale je kritizován za špatné ovládání (výběr vojáků) a chybějící multiplayer.

  Hry: Silent Storm: Sentinels, Night Watch, Day Watch, Heroes of Might and Magic

2 XNA

V této kapitole bych rád popsal framework XNA, který je jedním z prvních zástupců nástrojů, kde lze programovat DirectX v C#. Na začátku této kapitoly se zaměřuji na možnosti, které XNA nabízí. V závěru popisuji práci s efekty, jak je lze programovat nebo jak je můžeme propojit s XNA.

2.1 Představení XNA

XNA je nástroj, který si klade za cíl, zjednodušit programování počítačových her. Vznikl roku 2006 jako následovník Managed DirectX, poskytuje nám abstrakci nad knihovnou DirectX, díky které se o mnoho věcí nemusíme starat např. přístup k grafickému zařízení. Hry v XNA lze programovat v jazyce C#, který je jediný oficiálně podporovaný jazyk, ale lze využívat další jazyky podporující .NET. Pomocí těchto jazyků se vytváření her stalo mnohem jednodušší, tyto tzv. vysokoúrovňové jazyky přinesly do programování řadu změn, mezi které a nejvýznamnější patří garbage collector. Zařízení pro která lze hry vytvářet jsou počítače s Windows XP, Vista, 7, Xbox 360 a v posledních letech lze programovat také na Windows Phone 7, kde vývoj je téměř identický, ale nelze využívat všechny součástí XNA, jak pro PC nebo Xbox. Pro vývoj her potřebujeme mít nainstalované *Visual studio* a *XNA Game Studio Express*. XNA prošlo od roku 2006 různými změnami, vznikaly nové verze.

- Verze *XNA Game Studio 2.0* byla uvolněna 13. prosince 2007 pro Visual Studio 2005. Byla zavedena podpora pro Multiplayer a nastavování parametrů např. u modelů v content processoru.
- Verze *XNA Game Studio 3.0* verze byla uvolněna 5. října 2008 pro Visual Studio 2008, podporovala distribuci pro Xbox na Xbox live marketplace a vývoj pro Microsoft Zune.
- Verze *XNA Game Studio 3.1* byla uvolněna 11. června 2009 a přinesla např. podporu přehrávaní videa.
- Verze *XNA Game Studio 4.0* byla uvolněna 16. září 2010 pro Visual Studio 2010 a přinesla možnost vyvíjet pro Windows Phone 7 a přibyla možnost vybrat si ze dvou profilů *Reach* a *HiDef*.

V XNA 4.0 byly zavedeny profily *Reach* a *HiDef*. Profil *HiDef* je nadmnožinou profilu *Reach*, nabízí stejné možnosti jak profil *Reach*, ale má navíc některé funkce, proto je zapotřebí mít grafickou kartu s podporou DirectX 10 a výše. Pokud se pokusíme pustit hru v profilu *HiDef* na grafické kartě podporující pouze DirectX 9, tak neuspějeme. V tabulce můžeme pozorovat hlavní rozdíly mezi těmito profily [4].

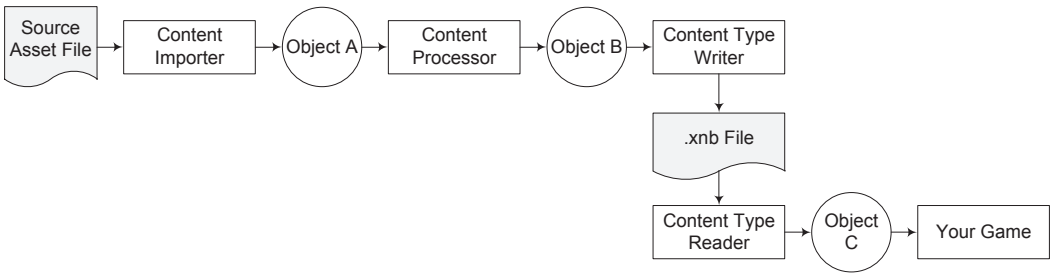
Tabulka 1: Porovnání profilů v XNA *Reach* a *HiDef*

	Reach	**HiDef**
Platformy	Windows Phone 7, Xbox 360, a Windows PC s DirectX 9	Xbox 360, Windows PC s DirectX 10 (nebo vyšší)
Shader Model	2.0 (Windows Phone nepodporuje vlastní shadery)	3.0 (Xbox 360 nabízí více možností)
Maximální velikost textury	2048	4096
Maximální velikost cubemap	512	4096
Maximální počet primitiv v jednom volání (Draw)	65535	1048575
Maximální počet Vertex Streamů	16	16
Maximální Vertex Stream Stride	25	255
Formát Index Bufferu	16 bit (short)	16 - 32 bit (short a int)
Vertex Element Format	Color, Byte4, Single, Vector2, Vector3, Vector4, Short2, Short4, NormalizedShort2, NormalizedShort4	Navíc HalfVector2, HalfVector4
Occlusion Queries	Nepodporuje	Podporuje

2.2 Načítání obsahu do XNA

XNA nabízí silný a jednoduchý nástroj pro načítání dat. *Content pipeline* slouží pro jednotné načítání jakéhokoliv z podporovaných formátů souboru. Jednoduchost spočívá v pouhém přetažení souboru do projektu *Content*. Při tomto způsobu načítání se nemusíme starat např. o jaký typ obrázku se jedná nebo zda načítáme model nebo texturu, vše je naprosto totožné. Po tomto přetažení se souboru přidělí tzv. *Asset name*. Toto jméno nám slouží pro identifikaci souboru a použijeme ho při volání metody `Load`, která ho přijímá jako parametr, vetšinou má stejné jméno jako název souboru. *Content pipeline* automatický rozpozná a jaký typ dat jde a přiřadí mu *content importer* a *content processor*. *Content pipeline* je omezen na načítaní a zpracování těchto souboru. Pro modely má *content pipeline* pouze dva typy souborů .X a .fbx. Pro načítání hudby můžeme zvolit soubory typu xap, mp3, wav a wma. Pro načítání videa je podpora pro typ wmv. Nachází se zde také velká podpora pro načítání různých typů obrázků .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, .tga. Dále obsahuje podporu také pro načítání fontů a efektů. Za načítaní obsahu je zodpovědná třída `ContentManager`.

Princip načítání obsahu do XNA



Obrázek 4: Pořadí volání jednotlivých tříd pro načtení herního obsahu

Princip:

- *Content importer* slouží jenom pro načítání souborů z média do operační paměti.Nic s těmito daty dále nedělá a pošle je např. ve stringu k dalšímu kroku.
- *Content processor* data přicházející z *content importeru* převezme zpracuje (rozparsuje) a převede je do nějaké struktury.
- Třída `ContentTypeWriter` převezme data z *content processoru* a uloží je do binárního souboru typu .XNB.

Tyto kroky se provedou jakmile daný projekt sestavíme. Při načítaní dat pomocí metody `Load` vlastnosti `Content` načte třída `ContentTypeReader` data z .XNB a předá je do proměnné, se kterou dále pracujeme. Pokud nám podporované formáty nebudou stačit můžeme si buď celou *content pipeline* napsat sami a nebo ji upravit. Veškeré třídy na obr. 4 jsou předpřipravené a stačí je dopsat.

2.3 Třídy XNA

V této podkapitole bych chtěl popsat základní třídy, ze kterých se skládá výsledná aplikace a jsou tedy velice důležité. V úvodu této podkapitoly se věnuji hlavní třídě `Game` a v závěru podobným třídám `GameComponent` a `DrawableGameComponent`.

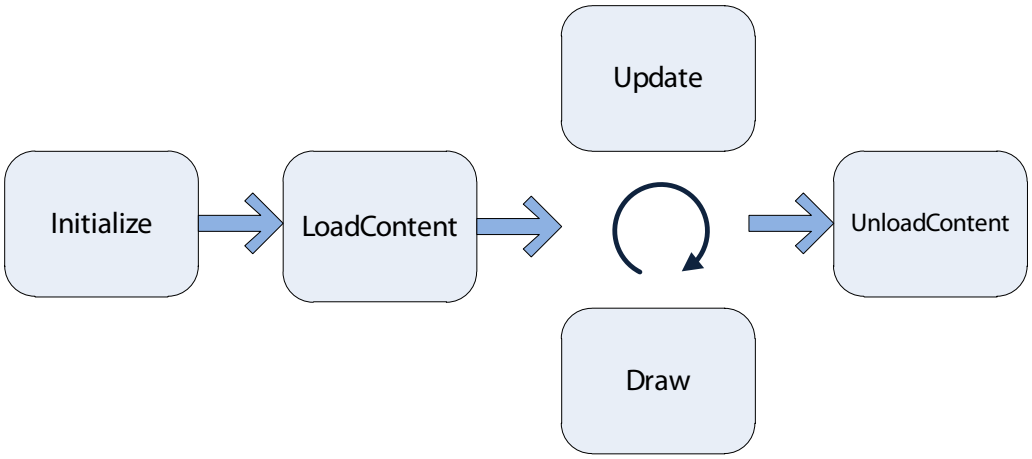
2.3.1 Třída Game

Hlavní třídou celého XNA je třídá `Game`. Tato třída má v sobě několik virtuálních metod, které je potřeba přepsat a pomocí metody `Run` se odstartuje jejich volání v daném pořadí. V konstruktoru této třídy se inicializuje třída `GraphicsDeviceManager`, díky které je možné pracovat s grafickou kartou, nastavit fullscreen, velikost a formát backbufferu, vertikální synchronizaci atd. Další metodou je `Initialize`, ve které inicializujeme všechny věci, které se netýkají grafické karty např. zvuk. Po této metodě se zavolá metoda `LoadContent`, kde načteme obsah z *content pipeline* pomoci vlastnosti `Content` a

její metody `Load`. Dále se v této třídě inicializuje objekt typu `SpriteBatch` pro vykreslování 2D grafiky. Dalšími a těmi nejdůležitějšími metodami jsou `Update` a `Draw`. Tyto metody se opakovaně volají po dobu běhu hry. Pokud máme zapnutou vertikální synchronizaci, tak počet volání je závislý na možnostech monitoru, při vypnuté vertikální synchronizaci je počet volání závislý na výkonu procesoru a grafické karty. V metodě `Update` řešíme hlavně uživatelské vstupy z klávesnice, myši a dalších zařízení. Dále také řešíme logiku jako je umístěni objektů (nastavování matic) nebo kolize. V metodě `Draw` provádíme vykreslování objektů.

2.3.2 Třídy GameComponent, DrawableGameComponent

Pokud vytváříme nějaké efekty např. skybox, lens flare a nechceme se starat o volání jejich metod, které můžou sloužit pro načtení grafického obsahu, logiku nebo vykreslení, můžeme v XNA vytvořit třídy, které budou dědit z tříd `DrawableGameComponent` nebo z `GameComponent`. `GameCopmonent` neobsahuje metody `LoadContent` a `Draw`, protože tato komponenta je hlavně pro řešení logiky. Třída `DrawableGameComponent`, která dědí od `GameComponent`, obsahuje všechny metody jako `Game` a lze s ní provádět vykreslování. Třída `Game` obsahuje vlastnost `Components` s metodou `Add`, kde tyto třídy vložíme a o nic se nemusíme starat, jejich metody se budou v daném pořadí automaticky volat. Tím můžeme ušetřit čas, ale hlavně náš kód bude přehlednější.



Obrázek 5: Pořadí volání metod ve frameworku XNA

2.4 Vstup

XNA také nabízí knihovnu *Microsoft.Xna.Framework.Input*, která v sobě obsahuje třídy, pro snadné používání vstupních zařízení, jako jsou klávesnice, myš a Xbox 360 Controller. Pro každé zařízení je určena třída obsahující metodu `GetState`, která vrátí aktuální stav konkrétního zařízení. Pro klávesnici zjišťujeme, zda jsme stiskli nějaké tlačítko pomocí metody `IsKeyDown`, která přijímá hodnotu určující tlačítko.

```
KeyBoard.GetState().IsKeyDown(Keys.Up);
```

Pro myš máme stejnou metodu jako u klávesnice `GetState`, která vrací aktuální stav tlačítek a navíc také pozici myši relativně od levého horního rohu okna. Na rozdíl od klávesnice poskytuje metodou `SetPosition` pro nastavení pozice myši.

Gametime

Vstupní zařízení nám nejčastěji určují pohyb ve scéně, ale každý má doma jiný počítač a je nutné, aby hra na každém počítači běžela stejně. Metoda `Update` a `Draw` má jako vstupní parametr třídu `GameTime`, která uchovává informace o čase a je nutné ji využít ke změně objektu ve hře, vynásobením hodnoty, která se stará o posun objektů, tímto časem. Kdybychom toho nevyužili hra by běžela na různých počítačích jinou rychlostí. Na rychlejších počítačích, kde by se metoda `Update` provedla $60\times$ za sekundu, objekt by se posunul o 60 jednotek, naproti tomu pokud by se metoda `Update` provedla $20\times$ za sekundu, objekt by se posunul pouze o 20 jednotek. Třída `GameTime` nám dává informace o čase, který uplynul od začátku spuštění hry nebo také kolik času uplynulo od předchozího snímku v různých jednotkách.

2.5 2D grafika

Pro vykreslování 2D grafiky je v XNA k dispozici objekt typu `Spritebatch`. Třída `Spritebatch` slouží jak pro vykreslování obrázků (textur) a textu, lze ji také využít k jiným účelům viz kapitola 3.3.

`Spritebatch` v sobě obsahuje tři důležité metody `Begin`, `Draw` a `End`. Metoda `Begin` slouží k přípravě grafiky na vykreslování. Tato metoda je přetížená a umožňuje nám nastavit důležité vlastnosti způsobu vykreslování. První parametr, který nastavujeme je `SpriteSortMode`, který nám určuje v jakém pořadí budou jednotlivé `Sprity` kresleny. Dalším parametrem `BlendState` můžeme nastavit vzájemné míchání jednotlivých `Spritů`. Parametr `SamplerState` nám určí, jakým způsobem budeme mapovat jednotlivé textury. Parametren `DepthStencilState` nastavíme možnosti hloubkové paměti. Poslední parametr `RasterizerState` nám určí, jak vektorová data budou převedena do rastrových dat. Metoda `Draw` přijímá parametry texturu, pozici, barvu, efekt nebo hloubku vrstvy, pokud je v metodě `Begin` nastavený parametr `SpriteSortMode` na hodnotu `Immediate`, provede se vykreslení ihned, jinak se čeká, až zavoláme metodu `End`, kde se provede vykreslení veškerých `Spritů` najednou [5].

```
spriteBatch.Begin();
spriteBatch.Draw(texture, new Rectangle(0, 0, 800, 480), Color.White);
spriteBatch.End();
```

2.6 Vertex a Index buffer

Vertex buffer je místo v paměti na grafické kartě, kde jsou uloženy informace o všech vertexech. Informace o vertex bufferu se zadávají do jeho konstruktoru, kde zadáváme o jaký

typ vrcholu se jedná, počet těchto vrcholů a zda se do tohoto bufferu bude pouze zapisovat. Data, se kterýma bude vertex buffer pracovat, naplníme pomocí metody `SetData`. V XNA jsou předvytvořeny struktury pro vrcholy, kterými vertex buffer naplníme.

- `VertexPositionColor` - informace o pozici a barvě
- `VertexPositionColorTexture` – informace o pozici, barvě a texturovacích souřadnicích
- `VertexPositionNormalTexture` - informace o pozici, normále a texturovacích souřadnicích
- `VertexPositionTexture` – informace o pozici a texturovacích souřadnicích

Pokud nám tyto struktury nebudou vyhovovat, lze si napsat jakoukoliv vlastní strukturu, která implementuje rozhraní `IVertexType` a vrací vlastnost `VertexDeclaration`, kde jsou obsaženy informace o vrcholech. Nutností je mít informaci alespoň o pozici vrcholu.

Index buffer je paměť, která na rozdíl od vertex bufferu obsahuje odkazy na jednotlivé vrcholy ve vertex bufferu. Do index bufferu nejčastěji tvoříme trojúhelníky v závislosti na nastavení po směru nebo proti směru hodinových ručiček. Výhodou použití index bufferu je, že grafická karta nemusí duplicitně transformovat vrcholy. Index buffer má také metodu `SetData`, kde vložíme pole typu `int` nebo `short`, obsahující odkazy na vrcholy ve vertex bufferu.

Kromě vertex a index bufferu má XNA také další dva buffery, které můžeme použít a dědí z tříd `IndexBufferu` a `VertexBufferu` a to `DynamicVertexBuffer` a `DynamicIndexBuffer`. Tyto buffery jsou optimalizovány pro práci, kdy chceme měnit vertex a index buffer každý snímek. Rozdíl rychlostí těchto bufferů je docela podstatný. Pro 262 144 vrcholů typu `VertexPositionNormalTexture`, kdy vrcholy jsou nastavovány do vertex a index bufferu každý snímek je výsledek pro klasický index a vertex buffer 15 snímků za vteřinu a pro dynamický index a vertex buffer je výsledek 30 snímku za vteřinu.

2.7 Vykreslení modelu

V XNA můžeme vykreslovat dva typy modelu .X a .fbx, aniž bychom museli napsat vlastní *content importer* a *content processor*. Model v XNA reprezentuje třída `Model`. Každý model se skládá z jedné či více dalších částí (sítí), které se v XNA nazývají `ModelMesh`. Každý `ModelMesh` může být rozdělen na více `ModelMeshPart`. Každá `ModelMeshPart` představuje jedno volání metody `Draw`, má kolekci trojúhelníků se stejným materiálem a `VertexDeclaration` (informace o vrcholech). My se musíme postarat o vykreslení všech sítí a nastavení potřebných informací např. *world*, *world*, *view*, *projection* matici. Jednou z důležitých věci je správně nastavit světovou matici. Třída `Model` obsahuje kolekci matic, které popisují pozici jednotlivých částí modelu, které jsou umístěny relativně k svému rodiči (další části modelu). S danou maticí je potřeba správně vynásobit část modelu tak, abychom dostali výslednou pozici.

```
foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (ModelMeshPart part in mesh.MeshParts)
        {
            BasicEffect effect = (BasicEffect)part.Effect;
            effect.World = transforms[mesh.ParentBone.Index];
            effect.View = view;
            effect.Projection = projection;
        }

        mesh.Draw();
    }
```

Lze využít také zkrácenou verzi pro vykreslení modelu, bez přístupu k `ModelMeshPart`.

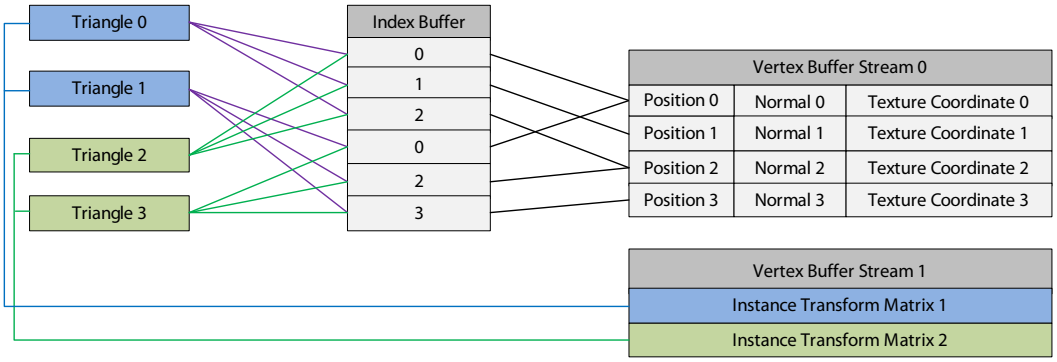
```
foreach (ModelMesh mesh in model.Meshes)
{

    foreach (BasicEffect effect in mesh.Effects)
    {
        effect.EnableDefaultLighting();
        effect.World = transforms[mesh.ParentBone.Index];
        effect.View = view;
        effect.Projection = projection;
    }

    mesh.Draw();
}
```

Vykreslování stejných modelů

Pro vykreslování více stejných objektů není vhodné vykreslovat každý objekt zvlášť a posílat GPU mnoho dat, proto bychom měli použít techniku, která ušetří práci CPU. Princip této techniky spočívá v nastavení dvou vertex bufferů metodou `SetVertexBuffers`, kde nastavujeme v prvním bufferu vrcholy vykreslovaného modelu a druhý buffer obsahuje světovou matici pro každou kopii objektu. Pro tuto matici je potřeba vytvořit speciální `VertexDeclaration`, kde každý řádek matice může zastupovat třeba `TEXCOORD`, výsledná `VertexDeclaration` bude mít tedy čtyři `TEXCOORD` pro každý řádek matice. Pro tuto techniku je potřeba si napsat vlastní shader verze 3.0, kde jako vstupní informaci do vertex shaderu přebíráme pozici modelu a světovou matici. Tímto dosáhneme např. pro 1000 objektů, místo $1000\times$ zavolaní metody `Draw` pouze jedno zavolaní. Stejná technika se také nachází v DirectX pod názvem *geometry instancing* a v OpenGL pod názvem *EXT_draw_instanced*.



Obrázek 6: Hardware instancing

2.8 Zvuk

Zvuk je důležitou součásti her. V XNA můžeme načíst zvuky typu wav, mp3 a další přes *content pipeline* do tříd typu `SoundEffect` nebo `Song`. Tyto třídy slouží pro přímé načtení zvuků přes *content pipeline*. `SoundEffect` je určený pro krátké zvukové efekty a třídu `Song` můžeme využít zase pro delší zvuky. Pro nahrávání zvuků můžeme využít také nástroj *XACT*, který je instalován spolu s XNA frameworkem. Jeho výsledný formát je taktéž podporován *content pipeline*. V tomto nástroji nejprve vytvoříme tzv. *wave bank*, který bude obsahovat všechny naše zvuky. Ve *wave bank* je možné nastavit vlastnost *in memory* nebo *streaming*, u vlastnosti *in memory* se nám zvuk načte celý do paměti, ze které je přehrávány a u *streaming* se vytvoří buffer, do které se nahraje vždy pouze určitá část. To slouží k tomu, abychom dlouhé zvuky nemuseli celé nahrávat do paměti, kde by zabíraly zbytečně místo. Dále se musí vytvořit *sound bank*, která nemá žádné fyzické zvuky, ale drží odkazy na zvuky v *wave bank* a určuje nám jak se zvuky z *wave bank* budou přehrávat. Další části je *cue*, což je jméno zvuku, které vidí programátor v XNA. Po uložení *XACT* projetku vznikne .xap soubor, který nahrajeme do naší *content pipeline*.

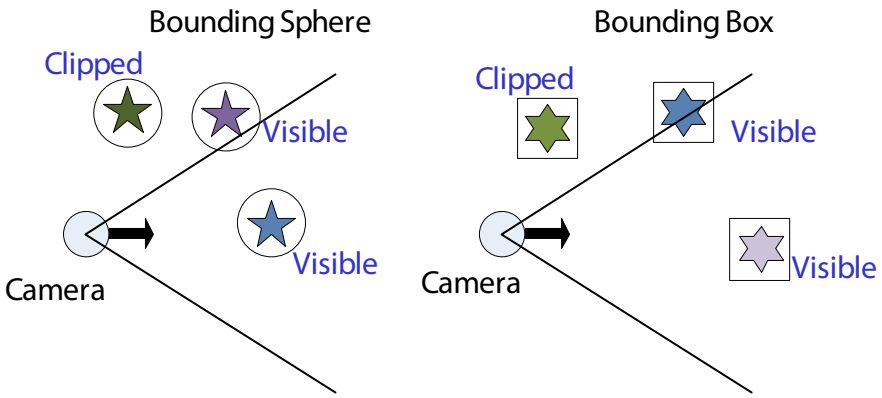
```
1  AudioEngine audioEngine = new AudioEngine("Content\\MyGameAudio.xgs");
2  WaveBank waveBank = new WaveBank(audioEngine, "Content\\Wave Bank.xwb");
3  SoundBank soundBank = new SoundBank(audioEngine, "Content\\Sound Bank.xsb");
4  soundBank.PlayCue("nazev");
```

2.9 Matematická knihovna

XNA také obsahuje matematickou knihovnu pro usnadnění práce. Máme možnost pracovat s třídami `Vector2`, `Vecto3`, `Vector4`, které nabízí mnoho metod např. pro skalární a vektorový součin, výpočet vektoru odrazu, transformaci vektoru maticí a mnoho dalších. Další třídou je `Matrix` pro práci s maticemi, umožňuje vytvářet *world*, *view* a *projection* matice, násobit je mezi sebou atd. Třídu `Quaternion` můžeme využívat například pro rotace. Třída `MathHelper` poskytuje různé konstanty např. π nebo také metody pro převod ze stupňů na radiány, lineární interpolace a další.

2.10 Obalová tělesa

Řešení kolizí je důležitou součástí každé hry. Kolize by se dali řešit více způsoby, ovšem jeden z nejrychlejších způsobů, jak toho dosáhnout, je obalit si jednotlivé modely do tvarů, jako jsou jehlan, koule nebo kvádr a testovat jejich kolize. Tento způsob testování není nejpřesnější pokud máme celý model obalený např. v kouli. V XNA máme jednotlivé modely rozdělené na jeden či více tzv. `ModelMesh`, kde každý `ModelMesh` může být obalený svým vlastním obalovým tělesem. Testování, zda několik objektů nekoliduje, budeme tedy nejprve provádět na celém modelu a pokud zjistíme kolizi, budeme testovat kolizi na jednotlivých částech modelu. Tím tento proces bude rychlý a zároveň o něco přesnější. V XNA máme předpřipraveno mnoho tříd pro obalování modelů, mezi které patří `BoundingBox`, `BoundingSphere` a `BoundingFrustum`, tyto třídy obsahují jak statické metody, díky kterým je možné sloučit více obalových těles do jednoho nebo vytvářet obalové těleso z jiných obalových těles, tak metody `Contains` nebo `Intersects` pro kontrolu, zda jedno těleso obsahuje nebo protíná druhé. Další ze tříd, které nám pomáhají v kolizích jsou `Plane` pro rovinu, a třída `Ray` pro paprsek, které mají metodu `Intersects`. Pokud načítáme model v XNA, tak každý model již má předpočítané obalové těleso `BoudingSphere`, které musíme pomocí světové matice transformovat, abychom mohli správně určit kolizi.



Obrázek 7: Kolize BoundingFrustum s BoundingBox a BoundingSphere

2.11 Transformace a matice

Transformace jsou důležitou součástí grafických aplikací a jsou často prováděnou matematickou operací v grafice. Transformace můžeme rozdělit na dva typy. První jsou transformace, které provádíme nad určitým objektem rotaci, translaci nebo změnu měřítka. A druhou transformací je transformace pro zobrazení objektů. Pro zobrazení 3D objektů na obrazovku je potřeba provést několik operací. Nejdříve musíme objekt, který má lokální souřadnice, převést do globálních souřadnic pomocí aplikace světové matice, která

se může skládat z matic translace, rotace nebo změny měřítka. Tím se dostaneme do tzv. *world space*. Další části je aplikace pohledové matice, která nám z *world space* udělá tzv. *view space*. A posledním krokem pro zobrazení 3D modelu na 2D obrazovku je vynásobení vrcholů projekční matici, čímž se dostaneme do tzv. *projection space*. Všechny tyto matice mají rozměr 4×4, kde čtvrtá hodnota představuje váhu bodu. Matici pro translaci zapíšeme jako:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{pmatrix}, \tag{1}$$

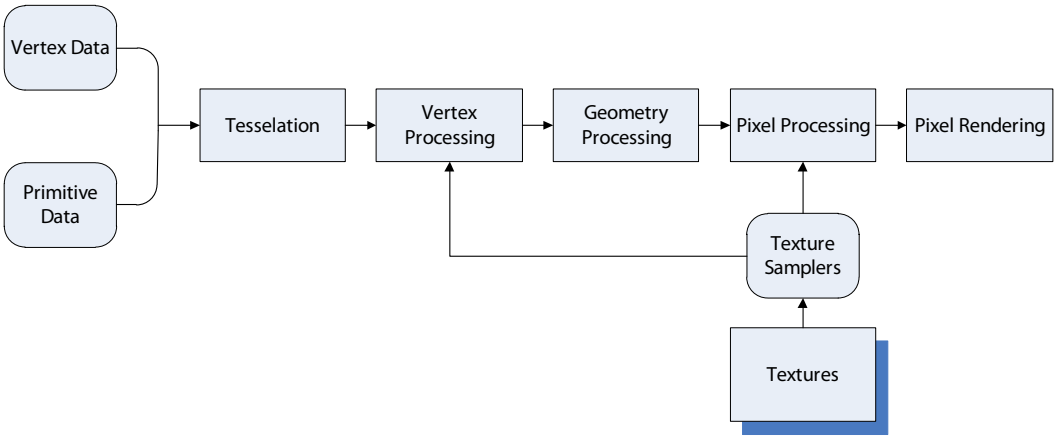
kde hodnoty pro translaci jsou na pozicích $M_{4,1}, M_{4,2}, M_{4,3}$. Matici pro rotaci kolem osy X zapíšeme jako:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{2}$$

kde hodnoty pro rotaci kolem osy X jsou na pozicích $M_{2,2}, M_{2,3}, M_{3,2}, M_{3,3}$, které jsou vypočítány pomocí goniometrických funkcí $\cos$, $\sin$ [6].

2.12 Effekty a HLSL

Pokud chceme v XNA něco vykreslit, potřebujeme nastavit efekt konkrétnímu modelu. XNA neobsahuje *fixed function pipeline*, který můžeme použít v DirectX nebo OpenGL a provádět s ním např. transformace, osvětlení a další. V XNA máme dvě možnosti přístupu, jak použít efekt na nějakém modelu. V prvním případě využijeme již předvytvořené efekty, kde nepotřebujeme znát žádný konkrétní jazyk a jejich použití spočívá v pouhém nastavování hodnot. Mezi tyto efekty patří *bassic efekt* pro transformace, osvětlení, mlhu a nanášení textur, dále také můžeme využít tyto efekty *dual texture efekt*, *skinned efekt* nebo *enviroment map efekt*. Využití těchto efektů je sice velice snadné, ale jejich použití nám přináší plno omezení. Proto máme možnost vytvářet své vlastní efekty. DirectX (XNA) má pro vytváření efektů svůj vlastní jazyk HLSL. Mezi další jazyky, které lze využít pro tvorbu efektu jsou GLSL pro OpenGL a CG od společnosti *NVIDIA* používaný pro OpenGL a DirectX. XNA nám zjednoduší práci oproti DirectX nebo SlimDX v načítání těchto efektu. Díky *content pipeline* můžeme efekty načítat jako další soubory v XNA.



Obrázek 8: Zobrazovací řetězec v *Direct3D*

Na začátku řetězce je blok *vertex data*. Tento blok obsahuje vrcholy, kde každý je reprezentován nějakou strukturou, která nese informace o tomto vrcholu. Tato informace může být pozice, normála, texturovací souřadnice, ale nejsme nijak omezení a můžeme vrcholu přiřadit jakoukoliv jinou informaci. *Vertex data* se ke zpracování předávají pomocí vertex bufferu, který je uložen na paměti grafické karty, ale lze je předat přímo z hlavní paměti, což může způsobit zpomalení. Další vstupní informací jsou *primitive data*. V tomto bloku vytváříme primitivy, kde jednotlivé indexy odkazují na data ve vertex bufferu. Mezi základní primitivy patří body, přímky nebo trojúhelníky. Tyto data předáváme ke zpracování pomocí index bufferu, uloženém na grafické kartě. *Tesselation* jednotka se stará o převod vyšších primitiv na základní primitiva. Blok *vertex processing* se stará o práci s vrcholy. Zde musíme vytvořit malý program tzv. *vertex shader*, který bude aplikován na všechny procházejí vrcholy stejným způsobem. Základní práci, kterou je nutné s těmito vrcholy udělat je transformace z *object space* do *screen space* nebo můžeme počítat např. osvětlení scény. Tento program lze naspat pomocí assembleru, ale pro pohodlnější práci je lepší zvolit jazyl HLSL. Po této části dochází k odstranění některých vrcholů. Odstraněny jsou vrcholy (trojúhelníky), které jsou dle našeho nastavení v XNA ve směru nebo proti směru hodinových ručiček. Dále jsou odstraněny vrcholy, které nejsou vidět. Po této části se dostáváme do fáze, kde se provádí rasterizace (převod primitiv na pixely). Po této fázi již nebudeme pracovat s vrcholy, ale s pixely. V bloku *pixel proccesing* stejně jako v bloku vertex proccesing, tak i zde musíme napsat krátký program tzv. *pixel shader* pomocí jazyku HLSL. Výstupem tohoto bloku je nejčastěji barva, která je v závislosti na mnoha parametrech vypočítaná. V tomto bloku se také provádí texturování, které také určuje výslednou barvu pixelu. Nakonec se provádějí různé testy, které určí výslednou scénu *(scissor test, alpha test, depth test, stencil test)*.

2.12.1 Programování efektů

Pro správné fungování efektu je nutné naprogramovat *vertex a pixel shader*. Ve výjimečných případech stačí pouze *pixel shader* (post proces efekty). *Vertex shader* je program, kde pracujeme s vrcholy (transformace, osvětleni...). S *vertex shaderem* můžeme pracovat s několika verzemi. Ve verzi *1* nemůžeme provádět klasické větvení pomocí instrukcí *if*, tato možnost byla dostupna až ve verzi *2*, kde se navyšoval počet instrukcí a registrů. Verze *3* je poslední, se kterou můžeme pracovat v XNA. Největší novinkou této verze je možnost získávat data z textury ve *vertex shaderu*. Pomocí programu *pixel shader* budeme pracovat s jednotlivými pixely provádět (per-pixel osvětlení, texturování...). *Pixel shader* v XNA je také stejně jako *vertex shader* v XNA použit maximálně do verze *3*.

2.12.2 HLSL

HLSL [7] je jazyk sloužící pro psaní efektu. Jeho syntaxe vychází z jazyka C. Oproti assembleru nám přináší řadu zjednodušení. HLSL má podporu jak základních datových typů (`bool`, `half`, `float`, `int`, `double`), tak obsahuje podporu vektorů (`float`, `int`, `double` (2, 3, 4)), matic (`float`, `int`, `double` (2x2, 3x3, 4x4)), struktur, textur, polí. Důležitou součásti je tzv. sémantika. Sémantika slouží k propojení dat mezi jednotlivými částmi efektu (vertex a pixel shader) a určuje nám k čemu daná proměnná slouží. Pro vstup do vertex shaderu můžem využít např. `POSITION`, `NORMAL`, `TANGENT`, `BINORMAL`, `TEXCOORD`. Pro výstup z vertex shaderu např. `COLOR`, `POSITION`, `FOG`, `TEXCOORD`. Pro vstup do pixel shaderu `COLOR`, `TEXCOORD`. Pro výstup z pixel shaderu `COLOR` a `DEPTH`. HLSL nám poskytuje také řadu funkcí, které můžeme využít např. `mul` pro násobení matic, `tex2D` pro získávání barvy z textury na základě texturovacích souřadnic atd. Dále také v každém efektu můžeme vytvářet několik technik, která zase může obsahovat několik průchodu.Toho můžeme využít např. pokud budeme používat více druhu osvětlení.

2.12.3 Propojení XNA s efekty

Vytvoření efektu provedeme v projektu *Content*, který stejně jako ostatní soubory načteme pomocí metody `Load`. K propojení dat z XNA do *vertex shaderu*, jak už bylo zmíněno slouží vertex buffer nebo přímo hlavní paměť. Ve vertex bufferu nastavujeme vlastnost `VertexDeclaration`, která definuje, jaký typ dat bude vrchol obsahovat nebo také nastavujeme počet těchto vrcholů. Pomocí metody `SetData` tyto data nastavíme. Nakonec je nutné nastavit z XNA proměnné, které budou stejné pro všechny vstupní data např. *world, view, projection* matice. To uděláme pomocí následujícího kódu.

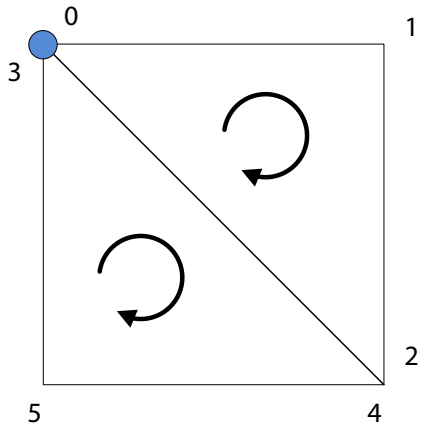
```
1 effect.Parameters["Promenna"].SetValue(hodnota)
```

3 Implementace

V této kapitole bych chtěl popsat jednotlivé části, ze kterých se skládá výsledná aplikace. Od implementace terénu pomocí level of detail, přes skybox, lensflare, post-proces efekty, vytvoření vody, tvorby stínu a problémy s nimi spojené až po částice.

3.1 Terén

Terén je jedním ze základních objektů, které se musejí nacházet ve scéně. Máme mnoho možností jak jej vytvořit. Jednou z možností je vytvořit terén v grafickém editoru a v XNA s ním pracovat jako s klasickým modelem, což může být mírně obtížnější. Dále se nabízí mnohem jednodušší možnost, vytvořit terén pomocí heightmap. Heightmap je dvourozměrná textura s různými úrovněmi šedi, kde bílá barva odpovídá nejvyššímu bodu terénu a černá analogicky opačně, každý pixel této textury zastupuje jeden vrchol. Nejjednodušší možností jak jej zobrazit, je pomocí tzv. *„Brute-force“* metody, kdy všechny vrcholy nahrajeme do vertex bufferu a v index bufferu z nich vytváříme trojúhelníky. Tato metoda nemusí být nejrychlejší viz kapitola 3.1.4.



Obrázek 9: Tvorba trojúhelníků v terénu

3.1.1 Normála

Pro každý vrchol je nutné vypočítat normálu, tedy kolmou přímku na daný vrchol. Výpočet normály bude záviset na výškách okolních bodů a lze jí vypočítat jako vektorový součin směrových vektorů od daného bodu k okolním bodům. Pro výpočet nám může stačit vektorový součin pouze dvou okolních bodů, ale zde by nebyla moc velká přesnost vypočítané normály. Proto je možné vypočítat vektorové součiny až s osmi okolními body a tím zajistit větší přesnost vypočítané normály. Výsledky vektorových součinů je nutné sečíst a následně normalizovat. Takto vypočítaná normála nám může sloužit pro

výpočet osvětlení, kde skalární součin vektoru světla s normálou nám určí, jaká intenzita světla bude dopadat na jednotlivé plochy terénu.

3.1.2 Určení sklonu vozidla

Normála nám nemusí pomáhat jenom pro výpočet osvětlení, ale také pro výpočet sklonu vozidla a výpočet světové matice (určení *Up*, *Forward* a *Right* vektoru).

Výpočet naklonění vozidla:

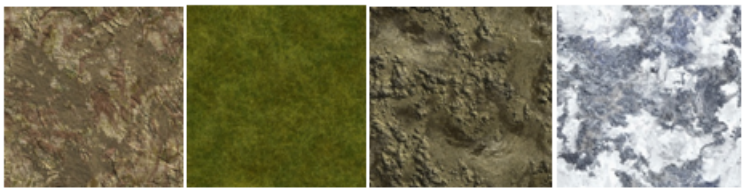
1. Před dopočtem světové matice je nutné udělat matici pozice a natočení vozidla, které spolu vynásobíme.
2. Vektor *Up* u světové matice vypočítané z prvního kroku nastavíme na normálu dané plochy, vypočítanou interpolací normál z okolních vrcholů.
3. Vektor *Right* světové matice vypočítáme jako vektorový součin vektoru *Forward* a normály.
4. Nakonec vypočítáme *Forward* vektor vektorovým součinu mezi normálou a *Right* vektorem světové matice.

3.1.3 Textura terénu

Pro každý vrchol musíme také určit texturovací souřadnice. Texturovací souřadnice určíme výpočtem:

$$U = \frac{x}{width}, V = \frac{y}{height}. \tag{3}$$

Tyto souřadnice by byli ovšem pro celý terén v rozsahu od 0 do 1. Terén při výšce a šířce 513 vrcholů by vypadal velice špatně, jedna textura by se roztáhla na celý terén, proto je nutné tyto souřadnice vynásobit daným číslem, abychom dostali lepší výsledek a na terén by se textura nanesla vícekrát. Takto vytvořený terén by ovšem byl pokryt pouze jednou texturou, proto v závislosti na výšce vypočteme, jakou texturu v dané výšce použijeme. Zvolil jsem čtyři textury, které se při navazování na sebe překrývají, aby výsledek lépe vypadal.



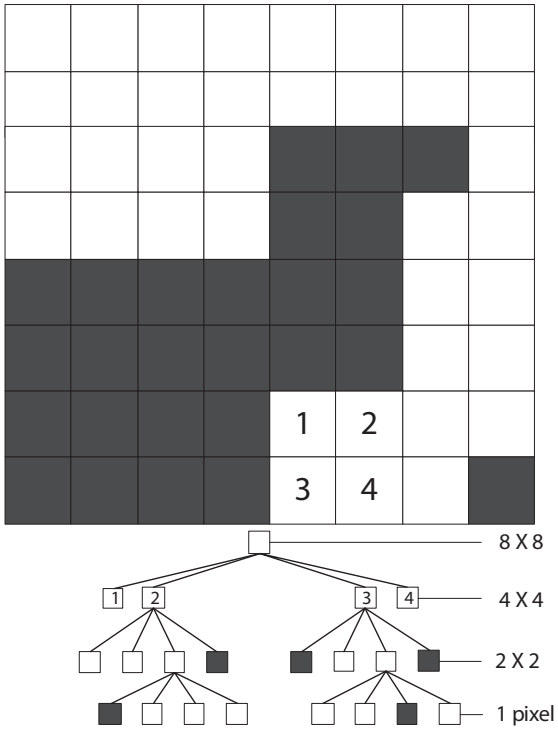
Obrázek 10: Textury terénu

3.1.4 Optimalizace terénu

Vykreslování terénu je velice náročná práce pro grafickou kartu. Pro terén o velikosti heightmap 513×513 je nutné vykreslit přes 260 000 vrcholů, proto je nutné vykreslování terénu optimalizovat. Pokud budeme vykreslovat terén o malé velikosti heightmapy není nutné optimalizaci provádět, avšak s rostoucí velikostí potřebujeme nějak zajistit, aby se nevykresloval celý terén. K tomu použijeme techniku *level of detail*, která si klade za cíl zmenšit složitost terénu. K tomu, abychom mohli implementovat *level of detail*, je potřeba si terén rozdělit pomocí struktury *quadtree*.

3.1.5 QuadTree

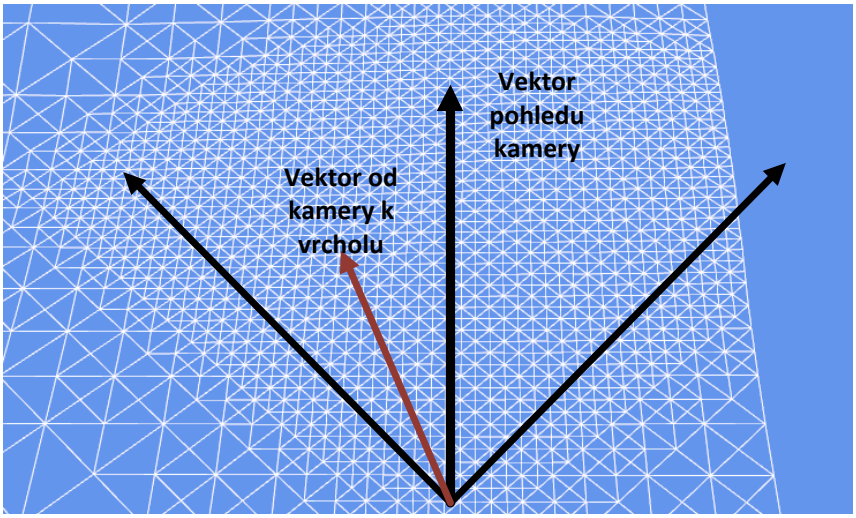
Quadtree je datová struktura, kde každý vnitřní uzel má právě čtyři děti. Používá se nejčastěji pro rekurzivní rozdělení dvourozměrného prostoru na stále menší čtverce nebo obdélníky. Kořen *quadtree* má největší velikost odpovídající celé oblasti. Nejmenší velikost má uzel nazvaný list, který nemá již žádné děti. Nejčastější užití *quadtree* je pro detekci kolizi nebo view frustum culling.



Obrázek 11: Způsob rozdělení terénu pomocí *quadtree* struktury

3.1.6 Level of detail

V mé práci, kde používám heightmap o veliskoti 513×513, jsem se rozhodl implementovat *level of detail* pomocí *quadTree*. Pro správné rozdělení terénu je nutné mít heightmap o velikosti $2^n + 1$, aby se každý region rozdělil přesně v polovině. Než začneme počítat vrcholy, které se nacházejí před námi, je nutné nejprve najít nejhlubší čtverec *quadtree*. Má první implementace spočívala ve vykreslování vrcholů, které se nacházely ve *view frustum*. Zde ovšem nastal problém při hledání nejhlubšího bodu, pokud kamera zabírala jenom část terénu. To mělo za následek nepříjemné problikávání terénu, způsobené v hledání nejhlubšího bodu, které se ne vždy podařilo najít. V druhé implementaci jsem se snažil vykreslovat vrcholy, kde vzdálenost vrcholu od kamery je pod určitou vzdáleností a aby nebylo nutné počítat vrcholy za kamerou, tak pomocí vektoru, který má směr pohledu kamery a směrového vektoru od kamery ke konkrétnímu vektoru spočítám, jaký je úhel mezi těmito vektory a zda je tento úhel pod určitou hranicí. V této implementaci nedocházelo k problikávání, protože nejhlubší bod, byl vždy nalezen a to i za cenu, kde terén nebyl vůbec ve vidět. Po hledání nejhlubšího čtverce kontrolujeme, zda se sousední čtverce nacházejí v daném okolí kamery tzn. zda splňují podmínku vzdálenosti od kamery a zda je úhel pod určitou velikostí. Pokud jsou tyto podmínky splněny, tak je označíme a z označených čtverců uděláme trojúhelníky, které následně vykreslíme.

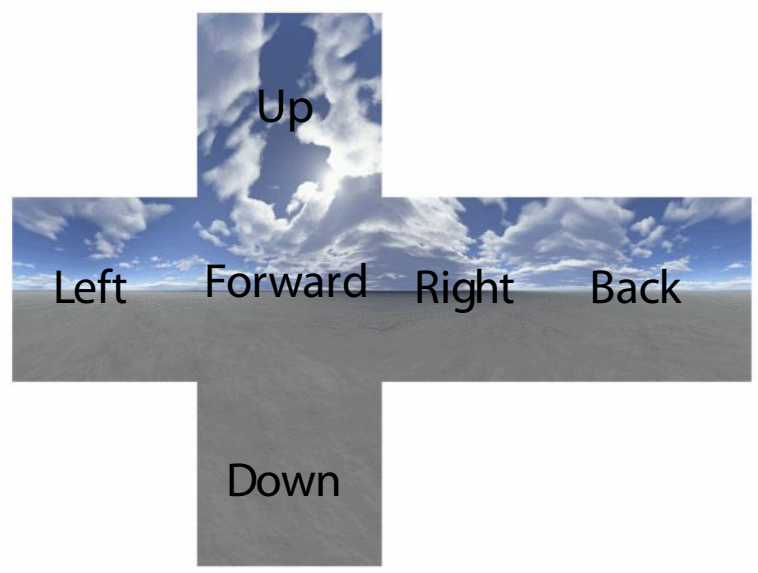


Obrázek 12: Level of detail terén. Skalární součin mezi vektorem kamery a vektorem od kamery k vrcholu nám určí zdá daný vrchol bude vykreslován.

3.2 Skybox

Důležitou součástí každé hry je obloha, která vytváří silnější dojem z každé hry. Pro implementaci a náročnost na GPU je skybox nejsnazší řešení vytvoření oblohy. Skybox je krychle, na kterou je nanesena textura. U těchto textur je důležité, aby na sebe navazovaly

a nešly vidět přechody mezi jednotlivými texturami. Implementace skyboxu spočívá ve vytvoření jednotkové krychle, která se bude neustále pohybovat s pozicí kamery. Před vykreslením skyboxu musíme vypnout zápis do hloubkové paměti a vykreslit ho jako první, tak se hloubka krychle nebude počítat s ostatními objekty a skybox se nám bude zdát nekonečně vzdálen.



Obrázek 13: Textura skybox

3.3 Postprocess efekt

Jednou z dalších důležitých součástí her, které dodávají na reálnosti, jsou tzv. postprocess efekty. Tyto efekty zpracovávají výstupní obraz vykreslené scény. Použití těchto efektů je v XNA velice snadné, celou scénu je potřeba vykreslit do textury a následně pomocí třídy `SpriteBatch` a aplikováním efektu (*pixel shaderu*), vykreslit na obrazovku. Existuje celá řada možných efektů, které lze tímto způsobem udělat např. *bloom efekt*, *sun-shafts efekt* a další, já jsem použil *wiggle efekt*, který nastává pokud se dostaneme pod vodní hladinu a *gaussian blur efekt*, který jsem použil při zásahu tanku.

3.3.1 Gaussian blur effekt

Gaussian blur efekt [8] způsobuje rozostření výsledného obrazu. Výsledný pixel se spočítá jako vážený průměr okolních pixelů jak ve vertikálním, tak horizontálním směru. Váha každého bodu je dána vzdáleností od aktuálního pixelu a je dána vzorcem:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y2}{2\sigma^2}}, \tag{4}$$

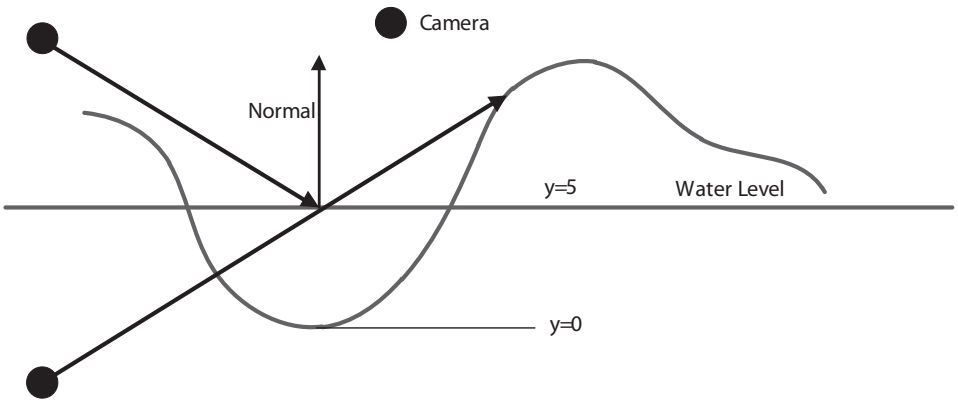
kde x je vzdálenost od aktuálního pixelu ve vertikálním tvaru, y vzdálenost v horizontálním tvaru a σ určuje jak moc bude obraz rozmazaný.

3.4 Voda

Existuje mnoho různých typů vod, které lze implementovat od složitých, typu oceánu, kde je potřeba mnoho vrcholů k vytvoření vln, až po ty nejjednodušší jako jsou jezera. Pro mou práci jsem zvolil ten nejjednodušší typ, kde je zapotřebí k implementaci pouze čtyř vrcholů a vytvoření vln bude jen iluzí.

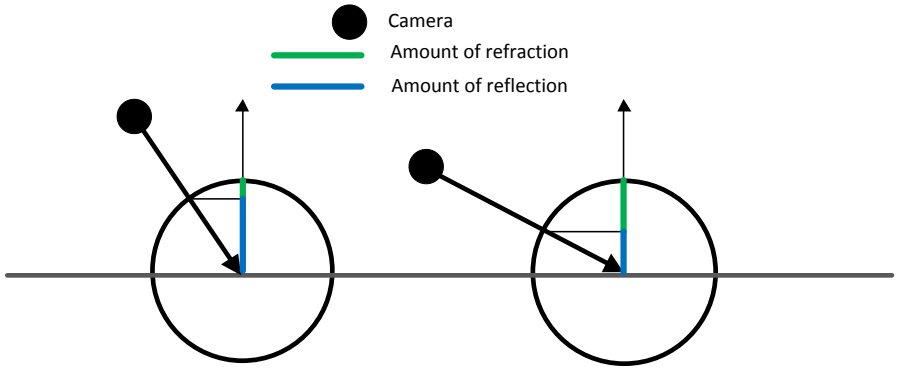
3.4.1 Reflection a refraction textura

Nejdůležitějším krokem k realizaci vody je vytvoření dvou textur *reflection* a *refraction*. *Refraction* textura nám vznikne vykreslením scény beze změny pozice kamery a cíle pohledu do textury, která bude obsahovat scénu nacházející se pod vodou. Aby textura obsahovala pouze scénu, která se nachází pouze pod vodou, nesmíme nad hladinou nic vykreslovat. Proto nastavíme ve výšce hladiny vody ořezávací rovinu a v pixel shaderu pomocí funkce `clip` tyto pixely odstraníme. *Reflection* texturu nám vytváří obraz ve vodě odražených objektů v závislosti na pozici pozorovatele. K vytvoření této textury musíme vytvořit kameru, která bude zrcadlem původní kamery podle výšky hladiny vody. Zde zase musíme vytvořit ořezávací rovinu, která bude odstraňovat pixely nacházející se pod hladinou vody.



Obrázek 14: Nastavení kamery pro *reflection* a *refraction* texturu

V závislosti na vektoru pohledu pozorovatele, který svírá s vektorem vodní hladiny určíme úhel, který bude udávat, jak hodně *reflection* a *refraction* textura bude ovlivňovat výslednou vodní hladinu. Lineární interpolaci určíme výslednou barvu. Pokud skalární součin pohledu pozorovatele s vodní hladinou bude 1 použijeme pouze *refraction* texturu, naopak pokud 0 použijeme jen *reflection* texturu.

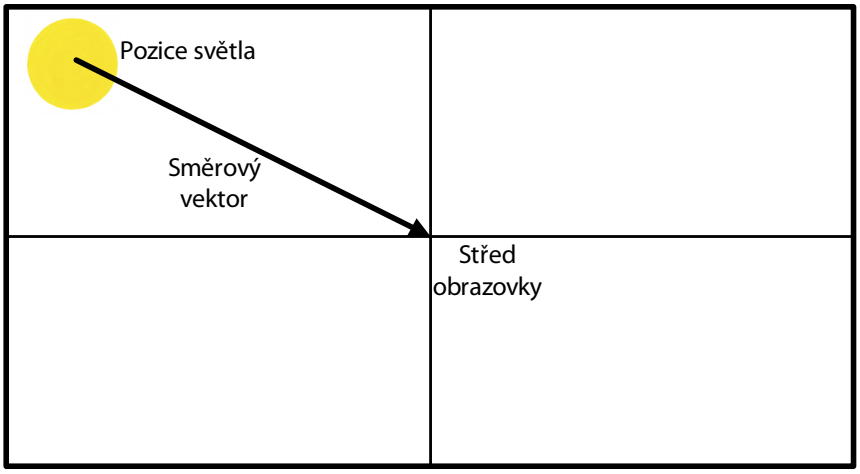


Obrázek 15: Princip využití *reflection* a *refraction* textur. Modrá barva nám určuje, jak hodně bude *refraction* textura ovlivňovat výslednou vodu. Naopak červená určuje využití *reflection* textury

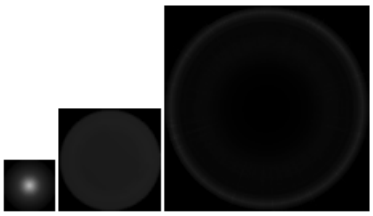
Na realizaci vytvoření iluze vln jsem použil normálovou mapu. Normálová mapa je textura, kde každý pixel obsahuje místo barvy normálu. K realizaci vln potřebujeme z této textury pouze dvě složky *(R, G)*, které budou přičteny k texturovacím souřadnicím a budou udávat, jak moc je voda v daném pixelu rozrušená. Dalším využití normálové mapy je výpočet odrazu slunce ve vodě, kde daná barva pixelu nám bude sloužit jako normála. Skalárním součinem mezi vypočteným vektorem odrazu a vektorem pohledu kamery určíme sílu odrazu slunce ve vodě [9].

3.5 Lens Flare

Lens flare [10] je optický efekt, který vzniká důsledkem interakce paprsků světla s čočkami kamery. Projevuje se jako mnohoúhelník nebo kruh, jeho tvar však závisí na tvaru membrány čočky. Lens flare může na jedné straně působit jako nechtěný artefakt, na druhé straně může vytvářet na fotografii nebo scéně lepší dojem. Realizace efektu není složitá, ve skutečnosti nám k tomu stačí vykreslit několik obrázků. Avšak problém nastává, když chceme určit zda světlo (světla) je viditelné kamerou.Zde musíme určit, zda světlu nestojí nic v cestě. Testování viditelnosti objektů kamerou lze provést pomocí třídy XNA `OcclusionQuery` [11], kde nejprve musíme vypnout zápis do *depth a color buferu*, ale je nutné mít povolený *depth test*. Pro zrychlení práce je nutné zároveň testovat více pixelů, proto budeme testovat určitý blok pixelů. Testování viditelnosti provádíme až na konci vykreslování, kdy už máme vykreslené všechny objekty, na základě hodnot uložených v *depth bufferu*. Test viditelnosti se provádí až v dalším snímku, kdy nám třída `OcclusionQuery` ve vlastnosti `PixelCount` vrátí počet viditelných pixelů. Počet těchto pixelů využijeme při zeslabování síly záře. Při vykreslování jednotlivých září musíme nejprve vypočítat vektor, po kterém budeme jednotlivé záře umístěné. Vektor bude mít směr od světelného zdroje do středu obrazovky. Pro výpočet umístění světla použijeme metodu `Project()` vlastnosti `ViewPort`, která nám převede umístění objektu ve *world space* do *screen space*, tedy souřadnice pixelu obrazovky.



Obrázek 16: Princip lens flare. Směrový vektor nám určuje umístění jednotlivých září



Obrázek 17: Lens flare textura

3.6 Stíny

Důležitou součásti každé hry jsou stíny. Dodávají každé hře nejen na realističnosti, ale také dávají informace o vzájemné poloze objektů. V dnešní době jsou nejpoužívanější dvě metody pro realizaci stínů. *Shadow volume* je metoda, kde pomocí siluety tělesa a *stencil bufferu* vytvoříme ostré stíny a *shadow map*, kde stíny vytvoříme pomocí stínové mapy. Tuto metodu jsem se rozhodl použít a vyzkoušet v XNA.

3.6.1 Shadow map

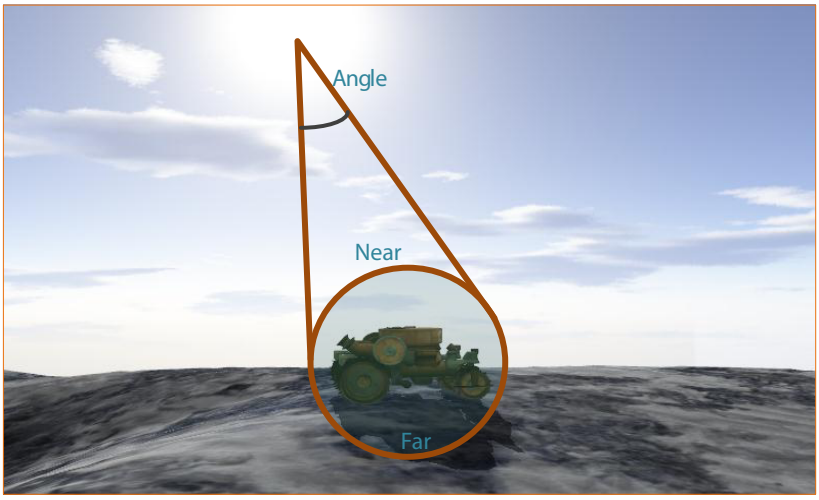
Postup:

1. Vytvoříme `RenderTarget`, do kterého vykreslíme místo barvy, vzdálenost od zdroje světla. Nejprve musíme natsavit pozici kamery na pozici světla a pomocí *pixel shaderu* zapsat do `RenderTargetu` vzdálenost.

2. Vykreslíme scénu z pohledu kamery. Zda se objekt se nachází ve stínu, určíme v *pixel shaderu*. Pozici zpracovávaného pixelu převedeme do souřadného systému světla pomocí daných matic. Hodnoty složek X a Y převedeného pixelu nám budou

sloužit pro získání vzdálenosti ze shadow mapy a složka Z nám určuje vzdálenost světla k danému pixelu. Porovnáním těchto hodnot určíme, zda se pixel nachází či nenachází ve stínu. Pokud je hodnota získána z shadow mapy menší než hodnota získána z pohledu světla pak se pixel nachází ve stínu. Pokud se pixel nenachází ve stínu nic s ním dále neděláme. Jinak hodnoty výsledné barvy vynásobíme danou hodnotou.

Problémem při vytváření shadow mapy je nastavit pohledovou a projekční matici, tak aby kamera zabírala pouze ty objekty, u kterých chceme, aby vytvářely stín. Tento problém můžeme v XNA vyřešit jednoduše pomoci obalových těles , kde u těch objektů, které mají vytvářet stín, obalíme daný objekt (objekty) do koule (`BoundingSphere`), pomocí které vytvoříme pohledovou a projekční matici.

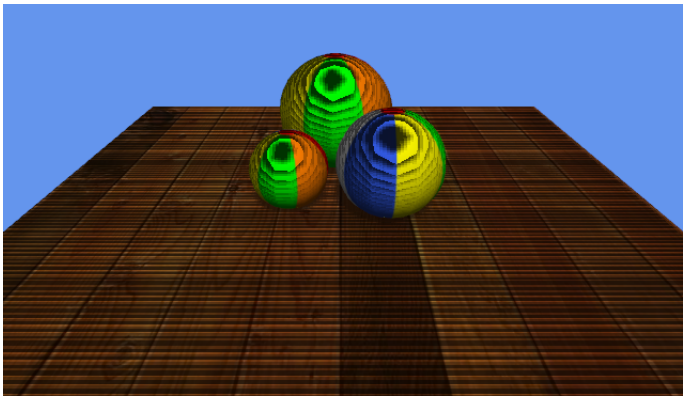


Obrázek 18: Princip tvorby pohledové a projekční matice. Kruh (koule) nám pomůže určit hodnoty pohledové a projekční matice (hodnoty near, far, angle a cíl pohledu)

Pro jednoduchost a hlavně rychlost jsem v mé práci zvolil stín, který vytváří pouze jeden objekt a tento objekt taktéž nemůže vytvářet stín sám na sebe. Tím jsem se zbavil mnoha problému, které mohou při vytváření stínu u techniky shadow map nastat. Mezi tyto dva nejdůležitější problémy patří *self shadow alias* a alias na obrysu stínu.

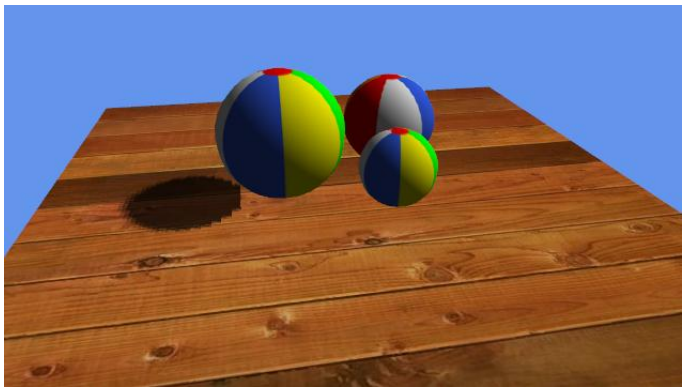
3.6.2 Self Shadow Alias

Self shadow alias nastává pokud objekt vytváří stín sám na sebe. Projevuje se tak, že plocha, která nemá být ve stínu se chová tak, jako kdyby byla částečně ve stínu. Problém nastává při vytváření stínové mapy, kdy při ukládání hodnot, se hodnota zaokrouhlí a tím při porovnání s hodnotou transformovanou do souřadnic světla vznikne nepřesnost. Tento problém můžeme vidět na obr. 19, kde přibližně 50% plochy je osvětlen a zbytek ve stínu.



Obrázek 19: Self-shadow alias.Na obrázku vidíme, jak asi 50% plochy, která nemá být ve stínu, se ve stínu nachází

Řešením tohoto problému je posunout hodnoty uložené ve stínové mapě o takovou hodnotu, aby tyto hodnoty byly dál než hodnoty získané z transformovaných souřadnic světla. Zvolená hodnota musí být dostatečně velká, aby tento problém odstranila, zároveň musíme dávat pozor, aby při velké hodnotě nedošlo k odstranění části stínu. Na obr. 20 lze tento problém vidět. Existuje mnoho technik, jak dosáhnout správné hodnoty, ale toto není cílem této práce.

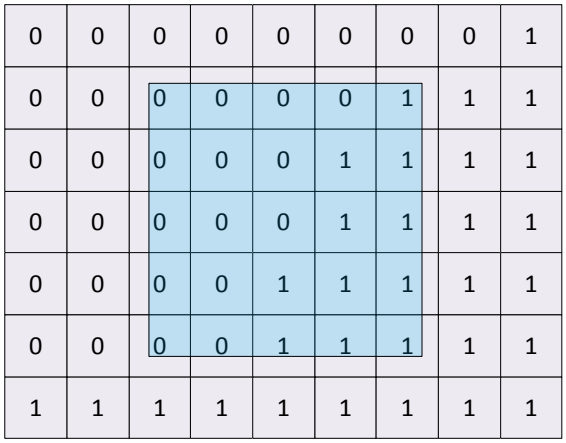


Obrázek 20: Velký posun hodnoty stínové mapy. Na obrázku lze pozorovat, jak stín postupně mizí

3.6.3 Alias obrysu stínu

Vytváření stínu je velmi náročnou operací, proto musíme stíny optimalizovat tak, aby jejich tvorba co nejméně zatěžovala grafickou kartu. Jedním z důležitých parametrů, které musíme správně nastavit je velikost textury, do které budeme vykreslovat hloubku od světla k danému pixelu. Pokud zvolíme velmi vysokou hodnotu (max. 4096), tak nám

razantně poklesne výkon, ovšem při malé zvolené hodnotě nedosáhneme takové kvality stínů a uvidíme zubaté okraje. Velikost této hodnoty závisí na tom, jak velkou scénu zobrazujeme, ale i při maximální velikosti textury budeme vidět na okrajích zubatý a ostrý obrys. Tento problém můžeme částečně vyřešit metodou *percentage-closer filtering* [12]. Tato metoda pracuje na základě hodnot získaných z okolí daného texelu a jejich zprůměrováním, tím dosáhneme na hranicích stínu určitého rozostření (získáme měkké stíny). Čím větší kvality chceme dosáhnout, tím volíme větší okolí daného texelu.



Obrázek 21: Princip PCF. Hodnoty 0 jsou texely, které nejsou ve stínu. Naopak hodnoty 1 se ve stínu nacházejí

3.7 Billboarding

Billboarding je technika, která se hlavně používá pro vzdálené objekty, u kterých není potřeba vykreslovat celý model, ale stačí vykreslit obdélník s daným obrázkem. Tato technika by měla významně zrychlit počítání celé scény i za cenu zhoršení kvality, protože pro vykreslení stromu stačí místo stovek vrcholů pouze čtyři. Billboarding není vhodný jenom pro vykreslování vzdálených objektů, ale také pro vykreslování částic. Cílem této techniky je zaručit, že čtverce nebo obdélníky budou vždy natočeny směrem ke kameře nebo k jinému objektu scény. K vytvoření Billboardingu je nutné naplnit světovou matici *Up*, *Right* a *Look* vektory.

Postup:

1. Nejprve je nutné vypočítat *Look* vektor, který má směr od daného objektu (čtverce, obdelníku) ke kameře. A následně normalizovat.

2. Dalším krokem je výpočet vektoru *Right*, který vypočteme vektorovým součinem mezi *Look* vektorem a vektorem *(0, 1, 0)*. Zde je nutné také vektor normalitovat.

3. Předposledním krokem je vypočítat *Up* vektor, který vznikne vektorovým součinem dvou předchozích vypočítaných vektorů.

4. Nakonec všemi vektory spolu s pozicí daného objektu naplníme světovou matici.

```
1    float3 Forward = normalize(Camera − Position);
2    float3 Right =normalize( cross(Forward, float3(0, 1, 0)));
3    float3 Up = cross(Right, Forward);
4    float3 Backward = −Forward;
5    float4x4 BillboardMatrix = {Right, 0, Up, 0, Backward, 0, Position, 1};
```

3.8 Částice

Důležitou součásti každé hry jsou částicové efekty. K základním částicových efektům patří kouř, oheň, výbuchy, sníh atd. Implementace těchto efektů může být založena na billboardingu, což je základní kámen částicových efektů, který nám umožní mít jednu částici jako obdélník natočený vždy ke kameře. Problémem při vykreslování částic je komunikace mezi CPU a GPU, kde na CPU se provádí výpočet simulace a na GPU provádíme billboarding, texturování a další nezbytné věci k vykreslení. Zde dochází ke značnému zpomalení, proto by bylo vhodné přenést veškeré výpočty na grafickou kartu a tím odstranit problém komunikace mezi CPU a GPU. V mé práci jsem implementoval částice, založené na billboardingu, kde výpočet probíhá na CPU.

Průběh vykreslování částic:

1. Do zadaného času budou neustále vznikat nové částice.

2. Po uplynutí dané doby začnou jednotlivé částice umírat.

3. Po vzniku všech částic se počká na dokončení všech živých částic a poté nastane konec vykreslování.

Obrázek 22: Textury použité pro vytvoření kouře a ohně

4 Fyzika ve hrách

Fyzika se v posledních letech stala ve hrách nepostradatelnou součástí. Fyzikální simulace se rok od roku stále více přibližuje realitě a díky tomu je potřeba mít výkonnější hardware. Dříve byl výpočet fyziky prováděn na CPU, ale výkon byl velice mizerný. Později firma Ageia přišla s nápadem počítat fyziku na specializovaném akcelerátoru s čipem PPU, ovšem tento pokus se při velkých cenách této karty a velmi malém zlepšení stal zbytečný a nepraktický. Další možností bylo počítat fyziku na GPU, kde je možné výpočty dobře paralelizovat, to se stalo velmi praktickým řešením a dnes je většina fyzikálních enginu počítaná právě na GPU.

V dnešní době, kdy je kladen důraz na rychlý vznik aplikací (her) se nevyplatí investovat čas do psaní svého vlastního fyzikálního enginu. Mnoho společností tyto enginy nabízí a mezi nejznámější enginy se řadí *PhysX* [13] nebo *Havok*. Obecně se fyzikální engine využívá na tyto věci.

- Výbuchy vytvářející prach a trosky.
- Postavy s propracovanou animací pohybu.
- Efekty realistických i sci-fi zbraní.
- Animace šatů, interakce textilní látky s dalšími objekty nebo větrem.
- Realistické chování mlhy nebo kouře i v případě složitých objektů, s nimiž přijdou do interakce (například puklina v zemi, postava procházející kouřem apod.).
- Simulace povětrnostních podmínek.
- Simulace chování kapalin.
- Animace částicových systémů.
- Umělá inteligence.

4.1 PhysX

PhysX je fyzikální engine vyroben firmou *Ageia*, ale od roku 2008 odkoupena firmou *NVIDIA*. V začátcích byla fyzika počítána na kartách PPU, kde software byl zdarma a peníze měl přinést prodej karet, ale díky špatnému výkonu se začalo počítání fyziky přenášet na GPU, kde za pomocí architektury CUDA probíhal výpočet paralelně, což přineslo značné urychlení počítání. Velkou nevýhodou PhysX je podpora pouze části grafických karet a to pouze grafický karet *GeForce*. Bez těchto karet je fyzika počítána na CPU a to přináší pokles na výkonu.

4.1.1 Začlenění PhysX do XNA

Pomocí fyzikálního enginu lze udělat z XNA ještě silnější nástroj k tvorbě her. Pro využití enginu v XNA je možné stáhnout PhysX.Net, který slouží jako wrapper z nativního PhysX, napsaný pomocí C++/CLI. PhysX je založen na tzv. *Actors*, kteří jsou hlavní protagonisté celé scény. *Actors* dělíme na statické a dynamické. U dynamických *actors* máme speciální vlastnosti, které musíme nastavit, aby se choval tak, jak reálné těleso. U statických *actors* tyto vlastnosti nenastavujeme, jak už z názvu poznáme, tak tyto *actors* by neměla vykonávat žádný pohyb, ale je možné také u těchto *actors* změnit pozici a další věci. PhysX však přepokládá, že statické *actors* by měli být opravdu statické a jsou proto optimalizované. Nějaké změny můžou způsobit nové výpočty a tím všechno zpomalit. PhysX nám dává možnost ručně měnit objekty pomocí nastavění těla na kinematické a za pomocí metody `MoveGlobalPoseTo` tyto tělesa přesouvat. Každý objekt může mít přiřazený nějaký tvar. Tento tvar je důležitý pro výpočet simulací. Objekt bez tvaru nemůže kolidovat s dalšími členy ve scéně. Mezi základní tvary patří koule nebo kvádr, ale můžeme využít trojúhelníkovou síť pro větší přesnost výpočtu. Každá actor má v sobě dvě vlastnosti `GlobalOrientation` a `GlobalPosition`, díky kterým jsem schopni vytvořit světovou matici a použít ji na náš objekt v XNA.

5 Porovnání s XNA

V této kapitole bych chtěl popsat, jaké výhody nebo naopak nevýhody nabízí framework XNA a další podobné nástroje. Na začátku kapitoly se zaměřuji na obecný popis frameworku, knihovny, enginu a jejich zástupců. V závěru popisuji, jak rychle jednotlivé nástroje pracují a jak rychle lze pomocí nich vytvořit jednoduchý terén.

5.1 Popis DirectX a SlimDx

DirectX

DirectX [14] je sada knihoven poskytující aplikační rozhraní, sloužící hlavně k vytváření her na OS Windows. Mezi základní komponenty patří *Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound*. DirectX má označovat souhrnný název pro všechny tyto části. Od roku 1995 bylo uvolněno přibližně 31 nových verzí. Poslední uvolněná verze je DirectX 11.1 pro Windows 8, která vkládá nové knihovny *DirectXMath, XAudio2 a XInput* převzaté z XNA.

Základní komponenty DirectX

- *Komponenta DirectDraw* využívaná pro práci s rastrovou 2D grafikou. Lze vytvářet aplikace v okně a v celoobrazovkovém režimu. Poskytuje také hardwarovou akceleraci.
- *Komponenta Direct3D* slouží pro práci s 3D grafikou. Poskytuje také hardwarovou akceleraci. K hlavním možnostem patří kreslení, mapování textur, stínování a další možnosti.
- *Komponenta DirectInput* poskytuje služby pro vstupní zařízení (klávesnice, myš, joystick, gamepade). Později nahrazen *XInput*.
- *Komponenta DirectSound* umožňuje komunikaci se zvukovými kartami.
- *Komponenta DirectMusic* slouží pro přehrávaní zvuků, hudby na vyšší úrovni než DirectSound. Má také podporu více formátů.
- *Komponenta DirectPlay* používaná pro síťové aplikace. Obsahuje implementaci modelu *TCP/IP*.

SlimDx

SlimDX [15] je open-source framework, který umožňuje využívat nativní DirectX v .NET pomocí programovacích jazyků C#, VB.NET nebo IronPython. K propojení mezi nativním DirectX a .NET je využit C++/CLI, který zde slouží jako most. SlimDx je postaven na knihovnách *Direct3D9, Direct3D10 a Direct3D11* a je možnost vyvíjet pouze na OS Windows. Poskytuje také třídy pro ulehčení práce např. `Math` pro práci s maticemi, vektory atd., třídy pro práci s obalovými tělesy, má také podporu *XACT* pro zvuky a další.

5.2 Porovnání framework, knihovna, engine

Framework vs Knihovna

Princip frameworku vystihuje tzv. *„Hollywood principle“* [16], který zní *„Don't call us, we'll call you“* , popisuje situaci, kdy se zájemci ucházejí o hraní ve filmu a na konec uslyší tuto větu. Princip frameworku je podobný. Framework je definován jako softwarová struktura, která má sloužit jako podpora při programování aplikací nabízející různé knihovny nebo návrhové vzory. Jinými slovy framework má již předvytvořené třídy, do kterých my doplňujeme kód definující logiku a tyto třídy se automatický v daném pořadí volají. U knihovny je situace opačná. Knihovna by se dala popsat, jako sada funkcí organizovaných do tříd. Rozdíl oproti frameworku je v tom, že my musíme vytvořit základní kostru tzn. funkce, do kterých budeme vkládat kód a ty pak jednotlivě volat. Programování her ve frameworku nám tedy nabízí pomocné konstrukce, díky kterým se programování ve frameworku může stát o něco lehčím, než programování her využitím pouze knihovny, která nám nabízí pouze určité funkce.

Framework vs Engine

Zda se budeme snažit vytvořit svou vlastní originální hru, kde nechceme být nijak omezování a nebo hru, kde využijeme již vytvořené části her. Před začátkem vývoje hry si musíme určit, jak naše hra bude vypadat nebo jaký cíl má splňovat. Zda se budeme snažit vytvořit svou vlastní originální hru, kde nechceme být nijak omezování a nebo hru, kde využijeme již vytvořené části her. A na tom bude záležet, který nástroj zvolíme. Výběrem frameworku budeme muset většinu věcí tvořit od začátku, což nám může vzít hodně času. Na druhou stranu nebudeme limitování žádným omezením, budeme moci vytvářet hry dle naší fantazie. Většina frameworků je také k dispozici zdarma. Pokud si zvolíme engine ušetříme mnoho práce a času.V enginu je většina části her jako je např. terén, skybox již připravená a lze je lehce využívat. Kvalitní engine nám také nabízí grafický editor, díky kterému nemusíme znát téměř žádný programovací jazyk. Nevýhodou enginu je, že se většinou omezuje pouze na malý okruh žánrů a za jejich nástroje při komerčním použití musíme zaplatit. Pro jednoduché a malé hry nemá cenu využívat nějaký engine a dobrou volbou je zvolit si framework, kde v případě potřeby si můžeme vybrat engine, který se zabývá pouze jednou části např. fyzika.

5.3 Porovnání XNA, DirectX, SlimDx

XNA vs DirectX

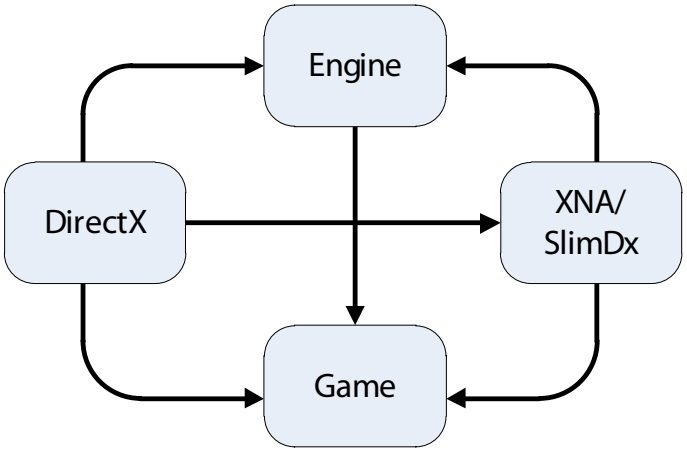
Rozdíl v programování her mezi knihovnou XNA a DirectX je docela podstatný. Před začátkem programování her si musíme určit, jaký cíl má naše hra splňovat. Zda je důležitější rychlost a využívání nových možností nebo jednoduchost a podpora více platforem. XNA přináší řadu zjednodušení, nad kterými bychom museli v nativním DirectX, strávit mnoho času (načítaní dat, vytváření herní smyčky . . .). Pokud bychom chtěli dělat hry pro jiné platformy jako WP7 nebo Xbox360, museli bychom zvolit XNA nebo engine.

Jako jeden z velkých problému používání DirectX, které mnoho lidí odradí, je používání jazyka C++, kde se o paměť musíme starat sami, což může přinést řadu problému a znepříjemnit nám vytváření her. Tento problém nenastane v XNA (C#), kde máme k dispozici automatickou správu paměti. Jako největší nevýhodu používání XNA je jeho omezení na *Direct3D9* a tím přicházet o nové možnosti v používaní dalších verzí DiretX. Jako vhodnou alternativu k XNA bych zvolil SlimDx, který z výhod DirectX přijímá podporu DirectX11 a nabízí automatickou správu paměti.

XNA vs SlimDX

Průkopníkem využívaní jazyka C# a .NET nad DirectX byl Managed DirectX. Tato knihovna byla představena roku 2002, postavena na .NET Framework 2.0 a podporovala *Direct3D9*. To přineslo oproti C/C++ řadu zjednodušení. Postupně byl vývoj této knihovny zastaven a nahrazen vývojem knihovny XNA. V dalších letech vznikly knihovny SlimDx a SharpDx, které se staly konkurenty pro XNA.

Výhody, které přináší používání nativního DirectX, jsou z velké části výhody také SlimDx. SlimDx je *„low-lewel wrapper“*, kde téměř všechny funkce z nativního DirectX lze najít také v SlimDx v identické podobě. Tento wrapper přinesl mnoha lidem zjednodušení práce, protože jsem zbaveni nutnosti používat ukazatelé (ruční správa paměti) a zároveň nemáme žádné omezení v používání nejnovějšího DirectX. Tím se SlimDx významně odlišuje od XNA, které má podporu pouze *Direct3D9*. Zároveň je nutné, aby vývojář ve SlimDx znal funkce nativního DirectX. Mezi další nevýhody SlimDx a zároveň výhody XNA patří nutnost napsat si vlastní načítaní souborů, což vidím jako největší problém SlimDx. SlimDx také neobsahuje třídu `SpriteBatch` pro 2D grafiku nebo třídy, které poskytují základní efekty např. *bassic efekt*, ovšem tyto dvě poslední nevýhody rozhodně nejsou důvodem, proč nepoužívat SlimDx.



Obrázek 23: Možnosti sestavení výsledné hry

5.4 Porovnání rychlostí nástrojů

Rychlost implementace v DirectX, XNA a Engine

Rychlost je důležitá vlastnost vývoje každé aplikace. Čím rychlejší a originálnější hra, tím má větší šanci uspět na trhu, kde je dneska obrovské množství různých her. Zvolením dobré technologie můžeme výrazně přispět k rychlejšímu vývoji. Porovnal jsem množství práce, které je nutné vyvinout při tvorbě a zobrazení terénu.

DirectX

- Základ je vytvoření okna. Okno můžeme vytvořit pomocí knihovny WinApi nebo Qt. Při vytváření okna pomocí *WinApi* nám funkce `CreateWindowEx` vrátí handle okna, kterým inicializujeme *Direct3D*. Důležitou věci je také napsat funkci, která se bude starat o obsluhu příchozích zpráv.
- Pomocí funkce `CreateDevice`, kde jeden z parametru je handle okna, vytvoříme objekt, pomocí kterého lze provádět práci s grafickým zařízením.
- Načteme texturu, ze které získáme data. Veškeré načítání probíhá ve zdrojovém kódu. Zároveň vytvoříme vertex a index buffer.
- Naplníme index a vertex bufferu daty z textury.
- Nastavíme scénu (vymažeme *render target a depth buffer*, nastavíme vertex a index buffer).
- Provedeme vykreslení.
- Nakonec musíme odstranit všechny objekty.

XNA

V XNA se o vytvoření okna, přístup ke grafické kartě a příjem zpráv postará třída Game. Zároveň se nemusíme starat o mazání objektů.

- Načteme texturu, ale oproti DirectX jí jenom přetáhneme a pomocí funkcí, které nabízí třída Texture2D získáme data.
- Vytvoříme a naplníme index a vertex bufferu daty z textury.
- Nastavíme scénu (vymažeme *render target a depth buffer*, nastavíme vertex a index buffer).
- Provedeme vykreslení.

CryEngine (Grafický editor)

- Načtení a další věci pro načtení a zobrazení terénu, lze udělat pomocí myši, bez větších znalostí.

Zde je vidět, že vykreslení terénu v DirectX může být práce na několik hodin, naproti tomu v enginu jsme schopni tutéž práci udělat během pár minut. Důležitý je také čas, který budeme věnovat studiu jednotlivých technologií. Pochopení základu programování v DirectX a jednoduché vykreslení modelu může zabrat několik dní až týdnů. Tento čas v XNA lze zkrátit na polovinu, kde nejsem nuceni vytvářet okno, získat přístup ke grafické kartě nebo načítat textury pomocí kódu. V *CryEnginu* se těmto věcem nemusíme vůbec věnovat a jsme schopni hned vykreslit terén.

Tabulka 2: Licence pro vybrané enginy

Engine	**Licence**
CryEngine	- zdarma pro nekomerční využití - komerční 20% z příjmů, bez přístupu ke zdrojovému kódu - možnost licence pro přístup ke zdrojovému kódu
Unreal Engine 3	- zdarma UDK pro nekomerční využití - komerční 25% z příjmů, bez přístupu ke zdrojovému kódu - licence pro přístup ke zdrojovému kódu za více než $700 000
Big World	- licence za $299 bez přístupu ke zdrojovému kódu, 10% z příjmů - licence za $2 999 s přístupem k zdrojovému kódu, 10% z příjmů

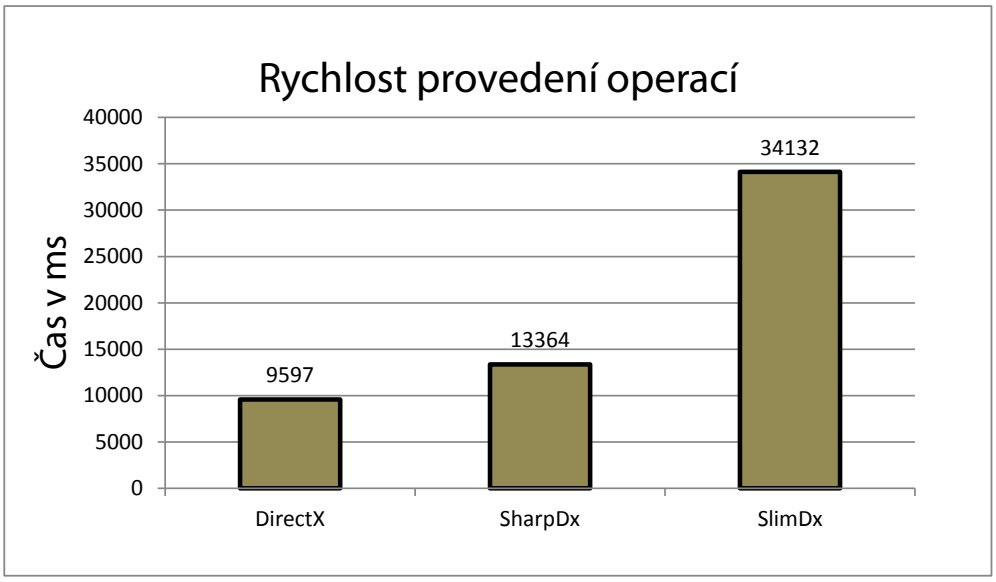
Rychlost SlimDx, SharpDx a DirectX

XNA, SlimDx nebo SharpDx přináší řadu zjednodušení ve vytváření grafických aplikací, ale toto zjednodušení má také stinnou stránku a to je rychlost. Hlavní příčinou je volání metod z nativního DirectX (C++) do jazyku s automatickou správou paměti (C#). Vyzkoušel jsem jakou cenu má toto volání na knihovnách SlimDx a SharpDx, které jsou jenom z velké části obaly metod z nativního DirectX a poskytují podporu DirectX11. Jelikož XNA tento DirectX nepodporuje, tudíž jsem ho v testu vynechal, ale výsledky by měli odpovídat SlimDx nebo SharpDx. V testu jsem se snažil nastavit všechny vlastnosti jak okna tak grafického zařízení na stejnou úroveň. Aby šli poznat nějaké rozdíly, tak test se skládá smyčky, která má milion průchodů a metody v ní nastaví *vertex buffer*, *vertex a pixel shader*, dále nastaví *rendertarget*, který vyčistí a vykreslí do něho trojúhelník.

```
for (int j = 0; j <= 1000000; j++)
{

        device->IASetInputLayout(layout);
        device ->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
        static UINT offset;
        device ->IASetVertexBuffers(0, 1, &vertices, (UINT*)&VERTEX_STRIDE, & offset);
        device ->VSSetShader(vertexShader, 0, 0);
        device ->RSSetViewports(1, &viewPort);
        device ->PSSetShader(pixelShader, 0, 0);
        device ->OMSetRenderTargets(1, & renderTarget, 0);
        device ->ClearRenderTargetView(renderTarget, blackColor);
        device ->Draw(3, 0);

}
```



Obrázek 24: Graf rychlostí jednotlivých nástrojů

Na grafu jsou vidět celkové časy po 1 000 000 průchodu cyklu v milisekundách. Také jsem změřil výsledky po 1000 průchodu, kde SlimDx měl v průměru *33 ms*, SharpDx *12 ms* a DirectX *8 ms*. Výsledky, které jsem naměřil, mají sice velké rozdíly mezi sebou, ale je třeba si uvědomit, abych dosáhl těchto výsledku, musel jsem tyto metody testovat v cyklu po 1 000 000 průchodů. Navíc rozdíl mezi SharpDx a DirectX je velice malý. Malý rozdíl mohl hrát také způsob měření.

6 Závěr

Cílem této práce nebylo vytvořit hru, která by se grafikou nebo fyzikou rovnala dnešním nejlepším hrám typu *Battlefield 3*, ale ukázat možnosti, které XNA framework nabízí. Mé krátké předchozí zkušenosti s knihovnami DirectX a OpenGL byly základem pro porovnání frameworku XNA s DirectX.

Tato práce mi udělala nový pohled na programování grafiky. Pokud začneme programovat grafiku s knihovnami, kde musíme začít psát programy od úplného základu, může být tato zkušenost odrazující a kvůli nízko-úrovňovému přístupu přináší mnohém více práce s jinými věcmi, jako je získání přístupu ke grafické kartě nebo načítání souborů než možnost soustředit se pouze na vytváření grafiky. Po vyzkoušení XNA mohu tedy z vlastní zkušenosti říct, že XNA tyto věci odstraňuje, ale způsob programování je podobný DirectX. Navíc tomu dopomáhá programovací jazyk C#, který spolu s *Visual studiem* vytváří kvalitní nástroj k tvorbě v XNA. Můžeme také využít kvalitní fórum, kde jsou vyřešeny téměř všechny problémy, které mohou nastat. Samozřejmě, že XNA má plno nevýhod, které nám naopak práci mohou přidělat, ale pokud se chceme naučit vytvářet grafiku, tak tento způsob bude pro nás nejlepší.

Další zkušeností, kterou jsem při psaní téhle práce dostal, byla možnost teoreticky a také trochu prakticky vyzkoušet práci s enginem, kde při porovnávání jsem zjistil, že pro vytváření her, kde se chceme starat pouze o grafiku se více hodí nějaký engine, který nám kvalitním editorem může odstranit práci v psaní her v některém programovacím jazyce a zároveň mnohem zrychluje vývoj.

Výsledná práce dává velkou možnost dále tuto hru rozšiřovat. Rozhodně by se hodilo do hry přidat více objektů, rozšířit nabídku zbraní, přidat více různých typu protivníků, zlepšit detekci kolize nebo částicové efekty, zapracovat na kvalitnějším zvuku a mnoho dalších věcí, které by tuto hru rozhodně zkvalitnilo.

7 Reference

[1] "Unreal Engine." http://www.unrealengine.com/.

[2] "CryEngine." http://www.unrealengine.com/.

[3] "Big World." http://www.bigworldtech.com/.

[4] Sean James, *3D Graphics with XNA Game Studio 4.0*. Packt Publishing Ltd., 2010. 275 s.

[5] Aaron Reed, *Learning XNA 4.0*. O'Reilly Media, 2010. 516 s.

[6] Eric Lengyel, *Mathematics for 3D Game Programming & Computer Graphics*. Charles River Media, 2001.

[7] Catalin Zima, "Crash Course in HLSL." http://www.catalinzima.com/tutorials/crash-course-in-hlsl/, 2011.

[8] Rafael C. Gonzalez, *Digital Image Processing*. Prentice Hall, 2002. 793 s.

[9] Riemer Grootjans, "The water technique." http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series4/The_water_technique.php, 2007.

[10] Alan Gordie, "Lens Flare Tutorial." http://archive.gamedev.net/archive/reference/articles/article813.html, 1999.

[11] Dean Sekulic, "Efficient occlusion culling." http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html, 2007.

[12] Michael Bunnell, Fabio Pellacini, "Shadow Map Antialiasing." http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html, 2007.

[13] NVIDIA, "PhysX." http://developer.nvidia.com/physx, 2012.

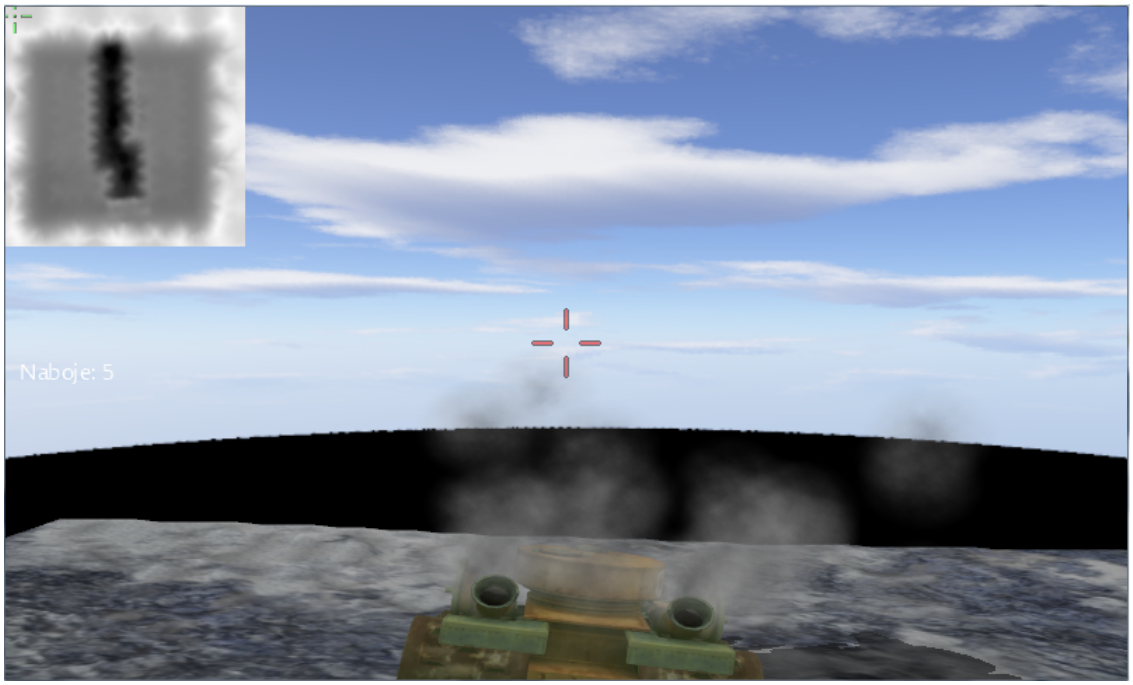
[14] Pavel Pokorný, *DirectX - Začínáme programovat*. Grada Publishing, 2008. 224 s.

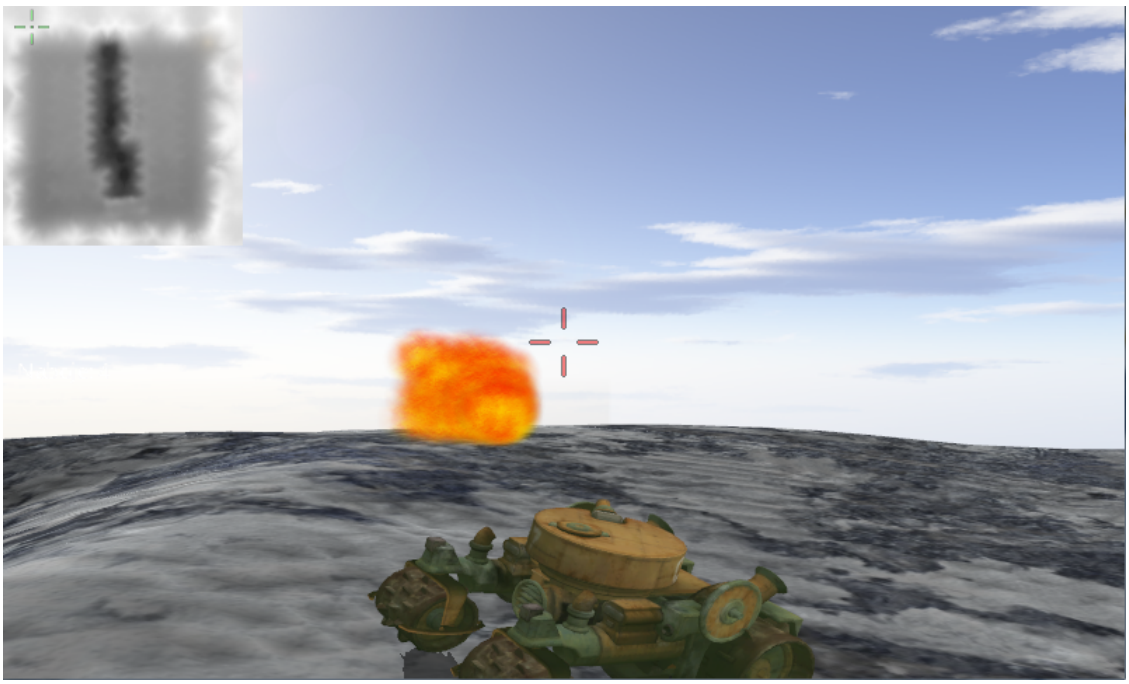
[15] SlimDX Group, "SlimDx." http://slimdx.org/.

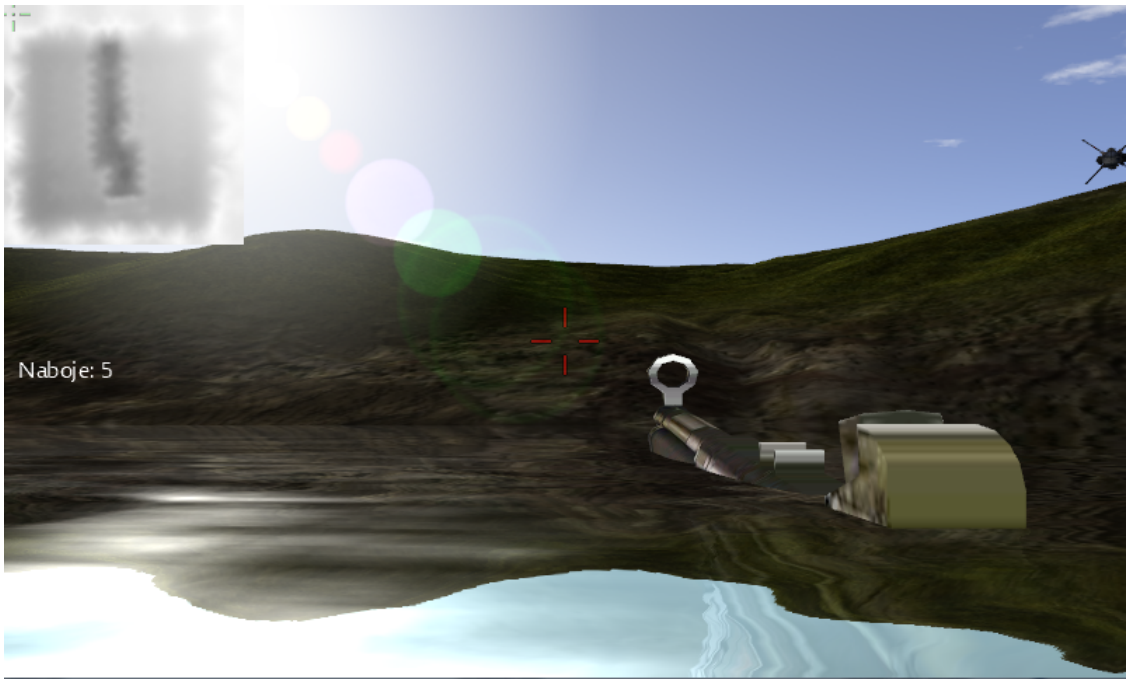
[16] Wikipedia, "Hollywood principle." http://en.wikipedia.org/wiki/Hollywood_principle, 2012.

[17] Jiří Žára, Bedřich Beneš, Jiří Sochor, Petr Felkel, *Moderní počítačová grafika*. Computer Press, 2005. 608 s.

A Obrázky ze hry






Naboje: 5